01 API Programming Fundamentals

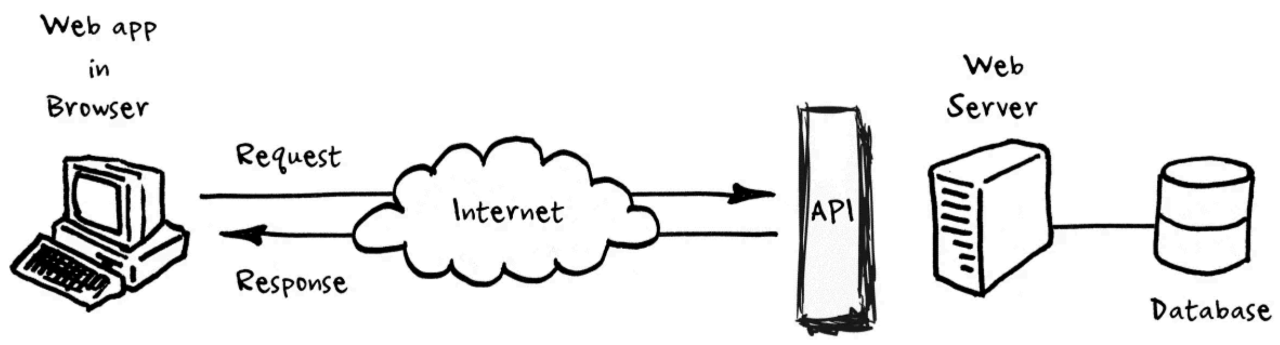
API Programming Fundamentals

API Fundamentals and Request Handling

What is an API?

> An API (Application Programming Interface) enables two pieces of software to communicate with each other. Think about the different ways you interface with software: You might open a mobile application to access your email. You might have a specific workflow to open messages, and file them away for later. Each of these workflows has a distinct "interface" or way in which you achieve a particular task.

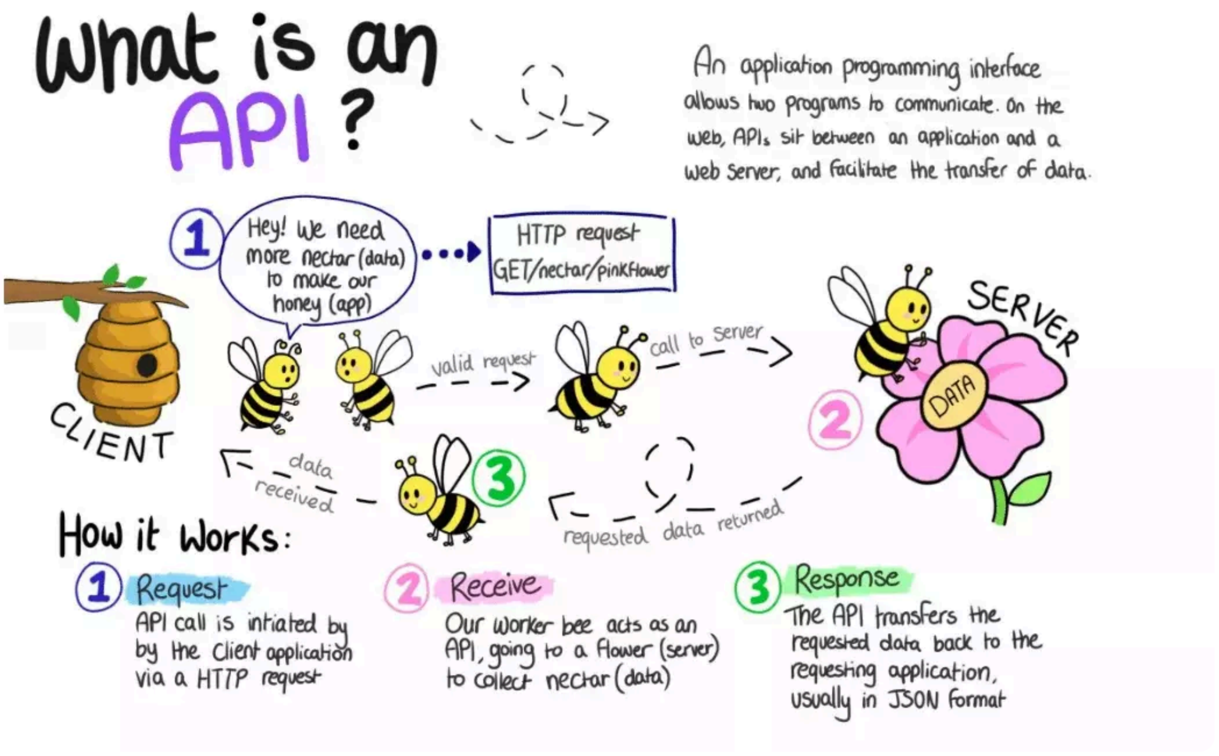
> An API is similar in concept. Instead of humans interfacing with software, the application interfaces with another application service. Rather than having a human point and click through a workflow, an API exposes functionality to another application.



An Application Programming Interface (API) is a set of rules and protocols that allows different software applications to communicate with each other. APIs define the methods and data formats that applications can use to request and exchange information.
Think of an API as a waiter in a restaurant:

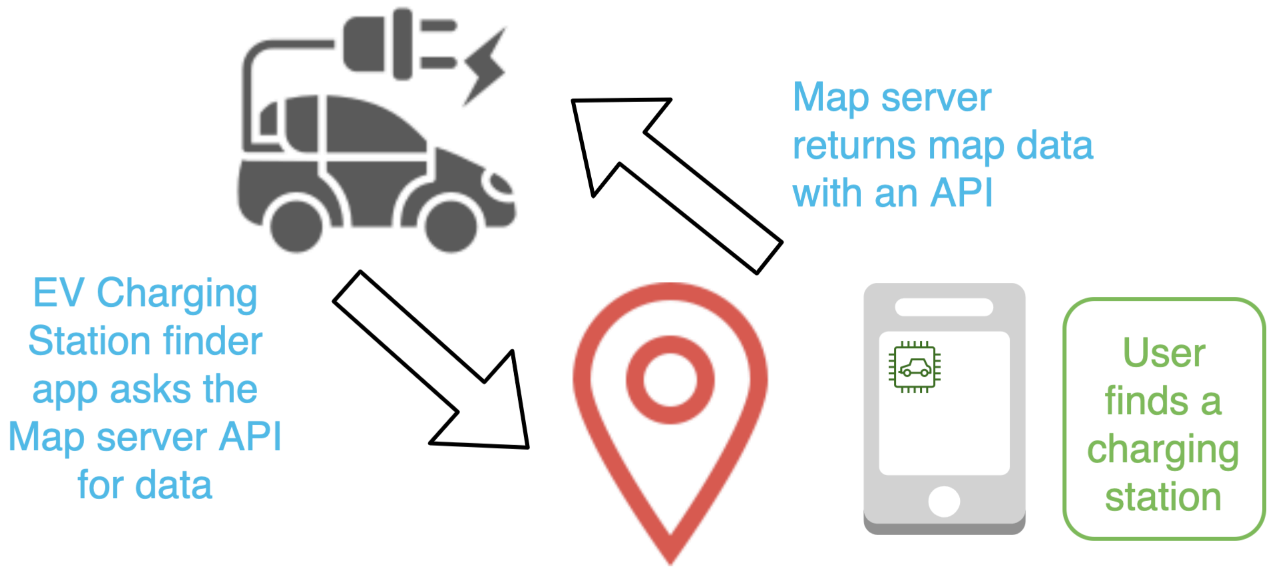
- You (the client) don't go directly into the kitchen (the server's internal systems)
- Instead, you give your order to the waiter (the API)
- The waiter takes your request to the kitchen and returns with your food (the data)

Another analogy is an electrical outlet: it provides a standard interface for devices to access power without needing to know the details of the power grid. APIs enable integration between systems, such as a weather app pulling data from a remote server or a payment gateway processing transactions in an e-commerce site.


What is an API?
An application programming interface allows two programs to communicate. On the web, APIs sit between an application and a web server, and facilitate the transfer of data.
1
Hey! We need more nectar (data) to make our honey (app)
HTTP request GET/nectar/pinkflower
CLIENT
valid request
call to server
SERVER
DATA
2
3
data received
requested data returned
How it Works:
1 Request
API call is intiated by by the Client application via a HTTP request
2 Receive
Our worker bee acts as an API, going to a flower (server) to collect nectar (data)
3 Response
The API transfers the requested data back to the requesting application, usually in JSON format

Creating APIs enables the development of rich applications with a wide range of functionality. For example, suppose you create a restaurant recommendation app. When users search for a restaurant, you want the app to return a list of relevant restaurants including a map of where they are. Would you create this functionality without existing data or some sort of starting framework? No, you look for interfaces for restaurant data and mapping data.

Creating that mapping capability might be slow, risky, and distract you from your core expertise, restaurants. Instead, you decide to use a third-party mapping capability. It's fast, inexpensive, and rock-solid. Now, your app can send a request to the map server's API and get map data in return.


Map server returns map data with an API
EV Charging Station finder app asks the Map server API for data
User finds a charging station

One role of an API is to act as a contract that enforces a specification. You want to know what to expect when you interface with a system.

Core API Concepts

APIs are fundamental building blocks in modern software development for several reasons:

1. **Abstraction**: APIs hide complex implementation details while exposing only necessary functionality, allowing developers to focus on high-level interactions.
2. **Reusability**: Well-designed APIs can be used across multiple applications, reducing development time and promoting consistency.
3. **Modularity**: Systems can be built as collections of independent services connected via APIs, enabling microservices architectures.
4. **Scalability**: API-based architectures can scale horizontally by adding more instances of services, with load balancing handled through intermediaries.
5. **Security**: APIs provide controlled access to resources, implementing authentication and authorization to protect sensitive data.
6. **Extensibility**: APIs can evolve over time with versioning, allowing backward compatibility while introducing new features.
7. **Interoperability**: APIs facilitate communication between diverse systems, languages, and platforms, fostering ecosystem growth (e.g., third-party integrations).

From practical perspectives, APIs drive innovation in areas like cloud computing, IoT, and mobile apps, where services like Google Maps API or Stripe payment API enable rapid development.

Types of APIs

1. **Web APIs**: Accessible over HTTP/HTTPS (REST, GraphQL, SOAP). These are the most common for internet-facing services, supporting formats like JSON or XML.
2. **Library/Framework APIs**: Programming interfaces provided by code libraries, such as Python's requests library for HTTP interactions.
3. **Operating System APIs**: Interfaces to interact with OS functionalities, like Windows API for file system access.
4. **Database APIs**: Interfaces to interact with databases (JDBC for Java, ODBC for general use), abstracting SQL queries.
5. **Hardware APIs**: Interfaces to interact with hardware components, such as WebUSB for browser-device communication.

In practice, web APIs dominate due to their platform-agnostic nature, but choosing the type depends on the use case—e.g., a mobile app might use a REST API for backend data while leveraging OS APIs for device sensors.

REST API Architecture

REST (Representational State Transfer) is an architectural style for designing networked applications. RESTful APIs have become the standard for web services due to their simplicity and scalability. Introduced by Roy Fielding in 2000, REST emphasizes resources as the key abstraction, where everything (e.g., a user, an order) is treated as a resource identified by a URI.

REST is an API framework that is built on HTTP, and the interface points are often for web services. When you combine REST and API, you see a simple service interface that enables applications or people to use the HTTP protocol to request objects or information.

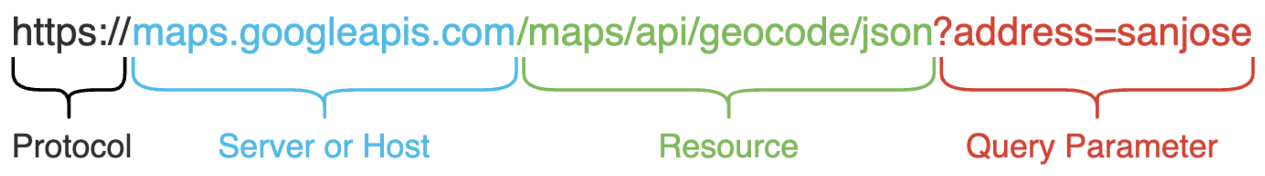
Key Principles of REST

1. **Stateless Communication**: Each request from client to server must contain all information needed to understand and process the request. The server doesn't store client context, improving

scalability but requiring clients to manage sessions.
2. **Client-Server Separation**: Client and server evolve independently, with the client handling UI and the server managing data storage. This enhances portability and scalability.
3. **Uniform Interface**: Standardized ways to interact with resources through:
   - Resource identification (URLs/URIs)
   - Resource manipulation through representations (e.g., JSON data)
   - Self-descriptive messages (including metadata like content type)
   - Hypermedia as the engine of application state (HATEOAS), where responses include links to related resources for navigation.
4. **Layered System**: Intermediary servers (e.g., proxies, gateways) can be used for load balancing, caching, or security without affecting client-server interaction.
5. **Cacheable**: Responses must define themselves as cacheable or non-cacheable, allowing clients or intermediaries to cache data for performance.
6. **Code on Demand (Optional)**: Servers can send executable code (e.g., JavaScript) to extend client functionality, though rarely used in practice.

Adhering to these principles makes APIs simple, lightweight, and fast. Best practices include maintaining consistency, using JSON for data exchange, and ensuring APIs are intuitive and minimal.

> REST APIs typically start with a URI. You can study the parts of this URI to determine what you are requesting.


https://maps.googleapis.com/maps/api/geocode/json?address=sanjose
Protocol
Server or Host
Resource
Query Parameter

Resources and URLs

In REST, everything is considered a resource, which can be:

- A document (e.g., an article)
- A service (e.g., "today's weather in New York")
- A collection of resources (e.g., all users)

Resources are identified by URLs (Uniform Resource Locators), following conventions like:

```
https://api.example.com/v1/resources/collection/item
```

Best practices for REST URLs:

- Use nouns, not verbs (e.g., `/users` not `/getUsers`)
- Use plural nouns for collections (e.g., `/users` not `/user`)
- Use hierarchical relationships (e.g., `/users/123/orders`)
- Include API versioning (e.g., `/v1/users`) to handle changes without breaking clients
- Maintain hierarchical structure with hyphens for word separation (e.g., `/user-profiles`), use lowercase, avoid special characters or file extensions
- Be consistent and intuitive for better developer experience

Practical tip: Use tools like Swagger for documenting URLs to improve usability.

HTTP Methods

HTTP defines several methods (also called "verbs") that indicate the desired action to be performed on a resource. These align with CRUD operations (Create, Read, Update, Delete).

Core HTTP Methods

1. **GET**: Retrieve data from a specified resource
   - Safe: Doesn't change server state
   - Idempotent: Multiple identical requests have same effect as a single request
   - Example: `GET /api/users/123` to fetch user details. In Python:

   ```
   import requests
   response = requests.get('https://jsonplaceholder.typicode.com/users/1')
   print(response.json())
   ```

2. **POST**: Submit data to create a new resource
   - Not safe: Changes server state
   - Not idempotent: Multiple identical requests create multiple resources
   - Example: `POST /api/users` with user data in request body. In Python:

```
import requests
data = {'name': 'John Doe', 'email': 'john@example.com'}
response = requests.post('https://jsonplaceholder.typicode.com/users', json=data)
print(response.json())
```

3. **PUT**: Update an existing resource (or create if it doesn't exist)
   - Not safe: Changes server state
   - Idempotent: Multiple identical requests have same effect as a single request
   - Example: `PUT /api/users/123` with updated user data. In Python:

   ```
   import requests
   data = {'name': 'Updated Name'}
   response = requests.put('https://jsonplaceholder.typicode.com/users/1',
   json=data)
   print(response.json())
   ```

4. **DELETE**: Remove a specified resource
   - Not safe: Changes server state
   - Idempotent: Multiple identical requests have same effect as a single request
   - Example: `DELETE /api/users/123`. In Python:

```python
import requests
response = requests.delete('https://jsonplaceholder.typicode.com/users/1')
print(response.status_code)  # Should be 200 or 204
```

Additional HTTP Methods

5. **PATCH**: Partially update a resource
   - Not safe: Changes server state
   - Not idempotent: Effects may vary
   - Example: `PATCH /api/users/123` with partial data. In Python:

   ```python
   import requests
   data = {'email': 'new@email.com'}
   response = requests.patch('https://jsonplaceholder.typicode.com/users/1',
   json=data)
   print(response.json())
   ```

6. **HEAD**: Same as GET but returns only HTTP headers, no body
   - Safe and idempotent
   - Useful for checking if a resource exists or has been modified (e.g., via ETag header)
7. **OPTIONS**: Returns supported HTTP methods for a specified URL
   - Safe and idempotent
   - Useful for CORS preflight requests to check allowed methods

Best practice: Use methods correctly to avoid complexity and ensure predictability.

Authentication Mechanisms

Authentication ensures that only authorized clients can access API resources. HTTP provides a framework for this, where servers challenge requests with a 401 Unauthorized response and a WWW-Authenticate header, and clients respond with an Authorization header containing credentials. Common methods include:

1. **Basic Authentication**: Encodes username:password in Base64. Simple but insecure without HTTPS, as it's easily decoded. Example header: `Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=`. Vulnerable to CSRF; use with TLS.
2. **API Key Authentication**: A unique key sent in headers (e.g., `X-API-Key: abc123`). Easy for identification but not secure alone; combine with HTTPS. Suitable for public APIs.
3. **Bearer Token (OAuth 2.0/JWT)**: Uses tokens for access, often JWTs for stateless auth. Example: `Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...`. Secure, scalable, and supports scopes/roles. OAuth 2.0 involves flows like authorization code for user consent.
4. **Digest Authentication**: Hashes credentials with a nonce to prevent replay attacks. More secure than Basic but less common now due to OAuth/JWT.
5. **Mutual Authentication**: Client and server authenticate each other, often via TLS certificates.
6. **Other Methods**: HOBA (HTTP Origin-Bound Authentication), NTLM, AWS-specific like AWS4-HMAC-SHA256.

Best practices: Use HTTPS always, prefer token-based methods for modern APIs, and implement rate limiting to prevent abuse. For setup in Python (using requests):

```python
import requests
from requests.auth import HTTPBasicAuth

response = requests.get('https://api.example.com/protected',
auth=HTTPBasicAuth('user', 'pass'))
print(response.json())
```

API Request/Response Cycle

Understanding the complete flow of an API request is essential for effective API development and troubleshooting. HTTP operates as a client-server protocol over TCP/IP, where clients fetch resources like HTML or JSON.

Request Components

1. **HTTP Method**: Defines the action (GET, POST, etc.)
2. **URL/Endpoint:** Identifies the resource (path from URL, excluding protocol/domain if inferred)
3. **Headers**: Metadata about the request (content type, authentication, etc.)
4. **Query Parameters:** Additional data passed in the URL (e.g., `?key=value&sort=asc`)
5. **Request Body**: Data sent to the server (typically JSON for modern APIs)
6. **Protocol Version**: E.g., HTTP/1.1 or HTTP/2 for multiplexing.

Response Components

1. **Status Code**: Numeric code indicating success/failure (200, 404, 500, etc.)
2. **Headers**: Metadata about the response (content type, caching directives, etc.)
3. **Response Body**: Data returned by the server (typically JSON)
4. **Protocol Version**: Matches the request.

Example Request/Response Cycle

1. Client prepares a request to create a new user:

   ```
   POST /api/users HTTP/1.1
   Host: api.example.com
   Content-Type: application/json
   Authorization: Bearer eyJhbGc...

   {
     "name": "John Doe",
     "email": "john@example.com"
   }
   ```

2. Request is sent over the network (TCP connection opened, potentially reused in HTTP/2).
3. Server receives and processes the request:

   - Validates authentication token
   - Parses JSON body
   - Validates input data
   - Creates user in database
   - Prepares response
4. Server sends response back to client:

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /api/users/456

{
  "id": 456,
  "name": "John Doe",
  "email": "john@example.com",
  "created_at": "2025-04-15T10:30:00Z"
}
```

5. Client processes the response:
   - Checks status code (201 means resource created)
   - Parses JSON response
   - Updates UI or takes next action

Proxies may intervene for caching or security. For efficiency, use persistent connections and compression.

HTTP Status Codes

Status codes are an important part of the API response. They provide immediate feedback on the result of a request and should be used consistently for better error handling.

Status Code Categories

1. **1xx (Informational)**: Request received, continuing process
   - 100: Continue
   - 101: Switching Protocols
2. **2xx (Success)**: Request successfully received, understood, and accepted
   - 200: OK - Standard success response (e.g., successful GET)
   - 201: Created - Resource created successfully (e.g., after POST)
   - 202: Accepted - Request accepted but processing not complete (async operations)
   - 204: No Content - Success but nothing to return (often used for DELETE or updates without data)
3. **3xx (Redirection)**: Further action needed to complete request
   - 301: Moved Permanently (permanent redirect)
   - 302: Found (Temporary Redirect)
   - 304: Not Modified (used with conditional GET requests for caching)

4. **4xx (Client Error):** Error caused by the client
   - 400: Bad Request - Generic client error (invalid syntax)
   - 401: Unauthorized - Authentication required or failed
   - 403: Forbidden - Authentication succeeded but user lacks permission
   - 404: Not Found - Resource doesn't exist
   - 405: Method Not Allowed - HTTP method not supported for this resource
   - 409: Conflict - Request conflicts with current state (e.g., duplicate resource)
   - 422: Unprocessable Entity - Request understood but semantically incorrect (validation errors)
5. **5xx (Server Error):** Error on the server side
   - 500: Internal Server Error - Generic server error (catch-all for unexpected issues)
   - 502: Bad Gateway - Server acting as gateway received invalid response
   - 503: Service Unavailable - Server temporarily unable to handle request (e.g., maintenance)
   - 504: Gateway Timeout - Gateway timeout waiting for response

Best practice: Include error details in the response body for 4xx/5xx codes, e.g., JSON with "message" and "details" for debugging.

🧾 Understanding HTTP Headers

🌐 What Are HTTP Headers?

HTTP headers are **key-value pairs** sent along with every HTTP request and response. They provide metadata about the request/response — such as content type, authentication, caching rules, user agent, etc.

They **don't contain the main content**, but rather **instructions or context** about the content. Headers are extensible, allowing custom ones prefixed with "X-".

🔥 Why Are Headers Important?

1. ✅ **Authentication & Security**
   Example: Pass tokens, cookies, or API keys to authenticate users.

`Authorization: Bearer <token>`

2. 📦 **Content Description & Negotiation**
   Tell the server what format you expect (like JSON, XML, HTML).

```
Accept: application/json
Content-Type: application/json
```

3. 🚀 **Custom Information**
   Some APIs use custom headers like:

```
X-User-ID: 123
X-Client-Version: 1.0.5
```

4. 📊 **Caching & Control**
   Control how responses are cached or revalidated.

```
Cache-Control: no-cache
If-None-Match: "abc123"
ETag: "abc123"  # For conditional requests
```

5. **CORS and Origin Control:** Headers like `Origin`, `Access-Control-Allow-Origin` manage cross-origin requests.

📦 Commonly Used Headers

Header	Purpose	Example Usage
`Authorization`	Sends credentials (token, basic auth)	Auth for protected endpoints
`Content-Type`	Declares format of request body	`application/json` for JSON data
`Accept`	Informs server what response format is expected	`application/xml` for XML
`User-Agent`	Identifies the client software	Browser or app version info
`Cache-Control`	Tells cache systems how to cache	`max-age=3600` for 1-hour caching
`X-*`	Custom headers used by APIs	`X-Rate-Limit: 100` for API limits
`Location`	Redirects or points to new resource	In 201 Created responses
`If-Modified-Since`	Conditional requests based on date	For caching efficiency

🤔 How to Know What Headers You Need?

- 📘 **Read the API Documentation** – The API usually tells you required headers.
- 🕵️ **Use Browser Dev Tools** – Open Dev Tools → Network tab → Inspect headers.
- 🔧 **API Tools like Postman or Insomnia** – They suggest required headers or add them automatically.
- 🧪 **Trial and Error** – Some APIs return helpful errors when headers are missing.

🐍 Example in Python (`httpx`)

```python
import httpx

headers = {
    "Authorization": "Bearer my_api_token",
    "Accept": "application/json",
    "User-Agent": "MyApp/1.0"
}

response = httpx.get("https://api.example.com/data", headers=headers)

print(response.status_code)
print(response.json())
```

📌 Bonus: Setting Headers with `requests`

```python
import requests

headers = {
    "User-Agent": "MyApp/1.0",
    "Accept": "application/json",
    "Content-Type": "application/json"
}

data = {"key": "value"}
response = requests.post("https://api.example.com/info", headers=headers,
json=data)
print(response.headers)  # View response headers
```

🚫 What Happens Without Headers?

- You may get a `401 Unauthorized` or `403 Forbidden` error.
- The server may respond in a different format than expected (e.g., HTML instead of JSON).
- Requests may be **rejected**, **delayed**, or **mishandled** (e.g., caching issues leading to stale data).

✅ Summary

- Headers are metadata for HTTP transactions.
- They are essential for **security**, **data format**, and **custom behavior**.
- Learn to read API docs and debug headers for success in web/API work. Best practices include using standard headers where possible and documenting custom ones.

Practical Labs for Hands-On Learning

To deepen understanding, perform these labs using tools like Postman (for GUI) or Python's requests library. Use the free fake REST API at https://jsonplaceholder.typicode.com/ for testing without a real server. This API simulates users, posts, etc., and supports all HTTP methods.

Lab 1: Exploring HTTP Methods (GET, POST, PUT, DELETE)

Objective: Practice core methods and observe responses.

1. **GET a Resource:**
   - In Postman: Send GET to https://jsonplaceholder.typicode.com/posts/1
   - Expected: 200 OK with JSON body.
   - In Python: Use the GET example above.
   - Analyze: Check headers like Content-Type.
2. **POST a New Resource:**
   - Body: `{"title": "My Post", "body": "Content", "userId": 1}`
   - Send POST to https://jsonplaceholder.typicode.com/posts
   - Expected: 201 Created with echoed data.
   - Note: Not persisted, but simulates creation.

3. **PUT to Update**:
   - Body: `{"id": 1, "title": "Updated Title"}`
   - Send PUT to https://jsonplaceholder.typicode.com/posts/1
   - Expected: 200 OK with updated data.
4. **DELETE a Resource**:
   - Send DELETE to https://jsonplaceholder.typicode.com/posts/1
   - Expected: 200 OK (or 204 No Content).

Discussion: Observe idempotency—repeat PUT/DELETE and see consistent results vs. POST creating "new" each time.

Lab 2: Authentication Mechanisms

Objective: Implement and test auth methods.

1. Use a public API requiring API keys, like OpenWeatherMap (sign up for free key at https://openweathermap.org/api).
   - Example: GET https://api.openweathermap.org/data/2.5/weather?q=London&appid=YOUR_API_KEY
   - In Postman: Add query param `appid` or header `X-Api-Key`.
   - Expected: 200 OK with weather data; without key: 401 Unauthorized.
2. Simulate Basic Auth:
   - Use https://httpbin.org/basic-auth/user/pass (public test server).
   - In Python: Use HTTPBasicAuth as in the example.
   - Observe: Without auth, 401; with correct, 200.
3. Explore Bearer Tokens:
   - Use a JWT playground like https://jwt.io/ to generate a token.
   - Test with https://httpbin.org/bearer (expects any token).

Discussion: Compare security—Basic is simple but weak; Bearer with JWT is more robust for production.

Lab 3: Request/Response Cycle and Headers

Objective: Inspect full cycle and manipulate headers.

1. Use Postman to send GET to https://jsonplaceholder.typicode.com/posts.
   - Add headers: Accept: application/json, Cache-Control: no-cache.
   - Inspect response: Status, headers (e.g., ETag for caching), body.
2. Conditional Request:
   - First GET a post, note ETag.
   - Second GET with If-None-Match: "ETag value".
   - Expected: 304 Not Modified if unchanged.
3. Error Handling:
   - Send invalid method (e.g., POST to a GET-only endpoint) for 405.
   - Analyze 4xx/5xx responses.

Discussion: Use browser DevTools to trace a real API call (e.g., to GitHub API) and note the cycle, including redirects or caching.

Lab 4: Building a Simple REST Client

Objective: Integrate concepts in code.

Write a Python script to interact with JSONPlaceholder:

```python
import requests

base_url = 'https://jsonplaceholder.typicode.com'

# GET all posts
response = requests.get(f'{base_url}/posts')
print('GET Status:', response.status_code)
print('First Post:', response.json()[0])

# POST new post
new_post = {'title': 'Lab Post', 'body': 'Content', 'userId': 1}
post_resp = requests.post(f'{base_url}/posts', json=new_post)
print('POST Status:', post_resp.status_code)
print('Created Post:', post_resp.json())

# PUT update
update = {'id': 1, 'title': 'Updated'}
put_resp = requests.put(f'{base_url}/posts/1', json=update)
print('PUT Status:', put_resp.status_code)

# DELETE
del_resp = requests.delete(f'{base_url}/posts/1')
print('DELETE Status:', del_resp.status_code)
```

Run and debug errors. Extend to handle auth if using a protected API.

These labs provide practical depth, reinforcing theory with real interactions. For advanced practice, build your own REST API using Flask or Express.js and test these concepts from the server side.