Web Scraping Using Python

UNIT-IV

Crawling Through API

Application Programming Interfaces (APIs)

An API (Application Programming Interface) is a set of definitions and protocols that allows two software components to communicate with each other. It's essentially a *contract* for interaction.

Analogy: The Restaurant Waiter

Think of a restaurant:

- **You (the Client):** The application requesting data or action.
- **The Kitchen (the Server):** The system that holds the resources and processes the requests.
- **The Waiter (the API):** The interface that takes your request (an order), translates it, delivers it to the kitchen, waits for the result (the meal), and brings the structured response back to you.

The API abstracts the complexity of the server's internal workings, allowing clients to interact with it using a simple, well-defined set of requests.

Key API Concepts (Web APIs)

Most APIs encountered in crawling are Web APIs, often following the REST (Representational State Transfer) architectural style:

- **Endpoint:** A specific URL where an API is hosted, representing a unique resource or function. (e.g., https://api.example.com/users/123).
- **HTTP Methods:** The "verbs" used to tell the server what to do with the resource.
    - **GET:** Retrieve data (e.g., get a user's details).
    - **POST:** Create a new resource (e.g., create a new user).
    - **PUT/PATCH:** Update an existing resource.
    - **DELETE:** Remove a resource.
- **Request & Response:** The client sends an HTTP Request to the endpoint, and the server returns an HTTP Response, typically containing a Status Code (e.g., 200 OK, 404 Not Found) and a Response Body (the data).

Parsing JSON

JSON (JavaScript Object Notation) is a lightweight, human-readable data interchange format. It's the most common format for data exchange in modern Web APIs, replacing older formats like XML.

JSON Structure

JSON is built on two structures:

1. A **collection of name/value pairs (Object):** Represented by curly braces (), similar to a Python dictionary or a JavaScript object.
   - *Example:* {"name": "Alice", "age": 30}
2. **An ordered list of values (Array):** Represented by square brackets ( ), similar to a Python list or a JavaScript array.
   - *Example:* ["apple", "banana", "cherry"]

What is Parsing JSON?

Parsing JSON is the process of converting the raw JSON text string received in an API response body into a native, usable data structure within a programming language.

- **Input:** A JSON-formatted string of text.
- **Process:** A JSON Parser (a library or built-in function like JSON.parse() in JavaScript or json.loads() in Python) reads the text, tokenizes it (breaks it into fundamental components like keys, values, and punctuation), and validates its syntax.
- **Output:** A language-specific data structure (e.g., a dictionary/map and list/array) that allows programmatic access to the data elements.

Theory of Parsing: The parser adheres to a formal grammar defined by the JSON standard. It performs lexical analysis (tokenizing) and syntactic analysis (checking the structure) to ensure the data is valid and can be mapped correctly to native data types (strings, numbers, booleans, arrays, and objects).

Undocumented APIs (Shadow APIs)

An Undocumented API is an Application Programming Interface, often a REST endpoint, that is used internally by a website or application but is not publicly advertised, officially supported, or listed in the provider's developer documentation. They are sometimes also referred to as Shadow APIs when created without the explicit knowledge of IT/security teams.

Why Do They Exist? (The Theory)

1. **Internal Use Only:** Developers build APIs for their own front-end applications (e.g., the website you browse). Since the front-end code is the only "consumer," there's no perceived need for public documentation.

2. **Legacy/Testing:** They might be old endpoints, test endpoints, or endpoints for features that were never fully released, remaining active but forgotten.
3. **Data Richness:** They often expose raw, structured data that is cleaner and richer than the information scraped from the main HTML page, which is why they are a prime target for automated data collection (crawling). They represent the purest form of the data being exchanged between the front-end and back-end.

Crawling Through APIs

Crawling, in this context, means making automated requests directly to these API endpoints rather than rendering and parsing the HTML of a webpage. This is usually faster, more reliable, and less resource-intensive because you get clean, structured JSON data immediately.

Finding Undocumented APIs

Finding these hidden interfaces is often a process of Reverse Engineering the communication between a website's client and its server.

1. **Browser Developer Tools (The Primary Method):**
    - Use a browser's Developer Console (F12 or Ctrl+Shift+I), specifically the Network Tab.
    - **Monitor Traffic:** Load the webpage and interact with the elements (e.g., click a button, scroll to load more, use a search bar).
    - **Filter Requests:** Filter the network log to show only XHR (XMLHttpRequest) or Fetch requests, as these are typically the asynchronous calls made by the client's JavaScript to the API.
    - **Inspect and Replicate:**
        - Examine the Request URL (the endpoint).
        - Examine the Request Method (GET, POST).
        - Examine the Request Headers (especially *Authorization* or *Cookie* headers needed for authentication).
        - Examine the Response Body (the JSON data) to confirm the endpoint is what you need.
2. **Analyzing Source Code:** Examining the website's JavaScript files can sometimes reveal API endpoints, parameters, and authentication logic directly embedded in the code.
3. **Network Proxies/Interceptors (Advanced):** Tools like Burp Suite or Fiddler sit between your browser and the internet, intercepting all traffic. They allow for deep

inspection and modification of requests/responses, which is helpful for complex authentication flows.

Documenting Undocumented APIs

Once an undocumented API is found, documenting it transforms it into a reusable, stable, and understandable data source. The documentation should capture the "contract" of the API.

Key Documentation Components

1. Endpoint URL and Method: The exact resource path and the HTTP method (e.g., GET /api/v1/products?limit=10).
2. Parameters/Payload: A description of all required and optional input, including URL query parameters (for GET requests) or the JSON body (for POST requests).
3. Authentication/Authorization: Details on how to authenticate (e.g., API keys, tokens in headers, cookies) to successfully access the data.
4. Response Structure: The exact structure of the JSON response, detailing the names and data types of all returned fields, including success and error responses.

Standardized Documentation: The industry standard for documenting REST APIs is the OpenAPI Specification (OAS) (formerly Swagger). An OAS document (usually in YAML or JSON format) provides a machine-readable specification of the entire API, which can then be used to generate human-readable documentation, SDKs, and automated tests.

Finding and Documenting APIs Automatically

Automating this process is crucial for managing large or frequently changing systems.

1. **Automated API Discovery (API Crawling):**
    - **Passive Monitoring:** Tools can monitor live network traffic (often at the API Gateway or Load Balancer level) to continuously identify every unique endpoint, method, and parameter being used by client applications.
    - **Code Analysis**: Static and dynamic analysis tools can scan application source code (especially JavaScript) to extract potential API URLs and parameters.
    - **Web Automation** (Headless Browsers): Scripts using tools like Selenium or Playwright can automatically simulate user actions (clicking, scrolling, form submissions) in a browser, while monitoring the Network tab to log all resulting API calls.
2. **Automated Documentation Generation:**
    - **Traffic-Based Generation**: Once API requests are discovered, specialized tools can inspect the request and response payloads, infer the data structures, and automatically generate a draft OpenAPI specification (OAS file). This is often

an iterative process where the tool observes traffic over time to build a robust model.

- **Schema Inference:** The tool uses the observed data to define the data model (schema) for both the request body and the response body, making an educated guess at field types (string, integer, boolean) and whether they are required.

This automatic process turns raw operational data into a structured API definition, greatly reducing the manual effort of managing API inventory and security.

Image Processing and Text Recognition

The core principle of text recognition is the signal-to-noise ratio. OCR engines like Tesseract are specialized pattern-matching systems. If the pattern (the character) is obscured by noise (blur, distortion, colour), the engine fails. Image preprocessing is the essential first step to maximize the signal (text clarity) and minimize the noise.

Fundamentals of Image Preprocessing for OCR

In digital image theory, an image is a two-dimensional array of pixels. Each pixel holds a numerical value representing colour and intensity.

1. **Gray scaling (Dimensionality Reduction):** Colour images use three channels (Red, Green, Blue—RGB). Gray scaling converts this three-dimensional data into a single-channel image, representing only intensity (luminance). This reduces computational load without losing the crucial information about character shape, as OCR primarily cares about contrast, not colour.
2. **Binarization (Dichotomization):** This is the process of converting a grayscale image into a binary image consisting of only two colours: black and white. It is achieved through thresholding.

Overview of Libraries and Pillow

Pillow (The Utility Knife)

The Pillow library (a maintained fork of the historical Python Imaging Library, or PIL) is the standard tool for handling images at a basic level. Its functions are rooted in raster graphics manipulation.

- Pillow excels at fundamental operations like opening, saving, resizing, rotating, and format conversion (e.g., converting a JPG to a TIFF, which is sometimes preferred by Tesseract).
- For preprocessing, Pillow's primary role is loading the image into a usable object and applying simple transformations like Image.convert('L') for grayscale and Image.point() for basic thresholding. It sets the foundation before passing the image

to more powerful computer vision libraries like OpenCV or directly to the OCR engine wrapper, Pytesseract.

Tesseract and Pytesseract (The Recognition Engine)

Tesseract is the OCR engine itself. Modern Tesseract (version 4 and up) uses a Long Short-Term Memory (LSTM) recurrent neural network for its character recognition stage. Pytesseract is simply the Python wrapper that calls the Tesseract executable, allowing the Python script to feed the image to Tesseract and receive the recognized text.

Processing Well-Formatted Text

Well-formatted text refers to clean documents, high-resolution scans, or clear printed pages. For such input, the OCR task is focused on speed and structural accuracy (retaining line breaks and paragraphs).

OCR Pipeline for Clean Text

The preprocessing steps for well-formatted text are minimal because the signal-to-noise ratio is already high.

1. **Deskewing:** The most crucial step is correcting any rotational misalignment (skew). The image is analyzed to detect the angle of the text lines, and a geometric transformation (rotation) is applied to make the text perfectly horizontal.
2. **Page Segmentation**: The OCR engine (Tesseract) employs internal Page Segmentation Modes (PSM). For a standard document, it uses PSM 3 ("Fully automatic page segmentation"), which attempts to recognize text blocks, lines, and words, respecting the layout.
3. **Lexical Correction (Post-Processing):** After initial character recognition, Tesseract uses an internal dictionary or lexicon (a word list) for the target language (e.g., English) to correct mistakes. If the engine sees a sequence of characters that is close to a dictionary word, it will adjust the recognized character to match the valid word, effectively using linguistic context to improve accuracy.

Reading CAPTCHAs and Training Tesseract

The challenge with CAPTCHAs is that they are an adversarial problem; they are specifically designed to break the fundamental assumption of OCR: clean segmentation.

CAPTCHA Preprocessing Challenges

CAPTCHA developers use techniques rooted in computer vision to deliberately degrade image quality:

- **Noise Injection:** Adding random dots, lines, or speckles to confuse the image's "blobs."

    - ***Solution:*** Median filtering or Morphological Operations (like erosion and dilation, which blur or thin features) are used to eliminate these small noise artifacts while preserving the larger characters.

- **Warping/Distortion:** Applying non-linear transformations (waves, arcs) to make the text non-uniform, which defeats Tesseract's built-in deskewing.
    - **Solution:** Image segmentation becomes necessary to try and isolate characters before distortion can be addressed.
- **Connected Characters:** Overlapping or touching characters (adhesion) make the critical step of segmentation (isolating one character from the next) impossible for general OCR.

The Theory of Fine-Tuning Tesseract (Transfer Learning)

For a specific, limited CAPTCHA pattern, generic Tesseract often fails. The solution is Fine-Tuning using the transfer learning concept from deep neural networks.

1. Ground Truth Collection: Thousands of CAPTCHA images must be generated and correctly labeled (the human-verified answer).
2. Training Data Generation: Tesseract tools convert the images and their labels into specific training files (.lstmf). Crucially, the process starts with a pre-trained model (the vast knowledge of English fonts, for example) as a foundation.
3. LSTM Retraining: The new CAPTCHA data is fed to the Tesseract LSTM engine with the instruction to continue training from the existing language model. This process modifies only the later layers of the neural network. This allows the model to quickly learn the specific distortions and fonts of the CAPTCHA while retaining its vast knowledge of the alphabet, leading to a highly accurate, domain-specific model (a custom .traineddata file).

Retrieving CAPTCHAs and Submitting Solutions

In an automated system, handling CAPTCHAs involves a controlled API interaction loop.

The Automated Loop

The process integrates image retrieval and text submission:

1. **Retrieval:** The automation script (often using the requests library or a headless browser like Selenium) makes an HTTP GET request to the CAPTCHA image source URL and downloads the binary image data.
2. **Solving:** The downloaded image is passed through the pre-processing steps and then to the trained Tesseract model to get the text solution.
3. **Submission:** The script then constructs a final POST request to the form submission endpoint, including all necessary form data *and* the OCR-solved text in the

designated CAPTCHA field. A successful submission confirms the block has been bypassed.

CAPTCHA Solving API Architecture

For modern, complex CAPTCHAs like reCAPTCHA (which relies on behavioral analysis and image selection), a third-party CAPTCHA Solving Service API is typically used, relying on human solvers or advanced AI/ML models.

- **Asynchronous Two-Step API:**
    1. **Task Creation (Input):** The crawler sends the CAPTCHA image data or the site's public key (e.g., the data-sitekey for reCAPTCHA) to the external service's API via a POST request. The service immediately responds with a unique Task ID.
    2. **Result Retrieval (Output):** The crawler, using the Task ID, repeatedly polls the service's API with GET requests until the human or AI has solved the challenge.
- **Token Submission**: For reCAPTCHA, the service doesn't return text; it returns a special token. This token is then inserted into the target website's hidden form field, completing the validation process. This method abstracts the complex, real-time solving process into a simple request-and-response API transaction.