INDIA WEB SCRAPER DOCUMENTATION

V 2.0

TEAM MEMBERS: ALICIA GITTENS & BETH DANIEL

# **Table of Contents**

|  |  |
| --- | --- |
| Overview | Pg 1 |
| Key Features | Pg 2-4 |
| Code Analysis | Pg 4-13 |
| Resources/Research | Pg-13 |

# **Overview**

This Python script is designed to help gather cybersecurity job listings from an online API, process the data to pull out useful details, and then save everything into a CSV file for easy viewing. When you run the script, it kicks off the main function, which goes through the steps of fetching the data, processing it, and saving it, all automatically. In the end, you get a CSV file filled with cybersecurity job listings, complete with all the important details neatly organized.

# **Key Features**

* **Data Extraction from an API:**
* **Pagination Support:** The extract\_data\_from\_api function fetches job listings from multiple pages (default is 5), allowing you to collect a larger dataset.
* **Error Handling:** It checks the HTTP response status code to ensure data is successfully retrieved, and prints an error message if not.
* **Comprehensive Data Transformation:**
* **Skills and Qualifications Extraction:**
  + The extract\_skills\_and\_qualifications function searches for skills, certifications, and qualifications across various possible keys in the job data.
  + It handles different data types (lists, dictionaries, strings) to gather as much relevant information as possible.
  + Uses a set to avoid duplicate entries.
* **Experience Extraction:**
  + The extract\_experience function improves upon basic extraction by handling nested dictionaries and checking job descriptions for mentions of "experience."
  + It constructs a readable experience requirement, combining years and descriptions if available.
* **Responsibilities Extraction:**
  + The extract\_responsibilities function scans the job description for sections related to duties and responsibilities.
  + It intelligently captures relevant lines while avoiding unrelated sections like requirements.
* **Job Status Determination:**
  + The extract\_job\_status function checks if the job is full-time, remote, contractor, or on-site.
  + It uses both explicit fields and keyword searches within the job description for accurate results.
* **Structured Data Compilation:**
* The transform\_job\_data function compiles all extracted information into a list of dictionaries, each representing a single job with standardized keys.
* It includes important fields like job title, company, location, date posted, skills, responsibilities, experience, job status, and application URL.
* **Data Loading and Exporting:**
* The load\_data\_to\_csv function utilizes pandas to convert the list of job dictionaries into a DataFrame.
* It exports the DataFrame to a CSV file, making the data easy to access and analyze in spreadsheet applications.
* **Modular Code Design:**
* The script is organized into separate functions for each major step (extract, transform, load), enhancing readability and maintainability.
* Helper functions are used to handle specific tasks, making the code reusable and easier to debug.
* **Timestamp Conversion:**
* Converts Unix timestamps to human-readable dates for the "Date Posted" field, improving the usability of the data.
* **Flexible Query Parameters:**
* Allows customization of the job search query and number of pages to retrieve, making it adaptable to different needs.
* The query can be modified to search for different job titles or platforms.
* **Error Messaging and User Feedback:**
* Provides clear console messages if data retrieval fails or if no job data is found, keeping the user informed about the script's progress.
* **Direct Execution Capability:**
* The main function enables the script to be run as a standalone program.
* Encapsulates the entire ETL process, from setting up API parameters to saving the CSV file.
* **Data Cleaning and Preparation:**
* Handles missing data gracefully by providing default values like 'N/A'.
* Strips whitespace and cleans up strings to ensure the final CSV is tidy and consistent.

# **Code Analysis**

**Original Code Analysis:**

**Code to Inspect JSON Response and Adjust CSV Export**

import requests

import json

import csv

# API request setup

url = "<https://jsearch.p.rapidapi.com/search>"

querystring = {"query": "cybersecurity jobs on glassdoor", "page": "1", "num\_pages": "1", "date\_posted": "all"}

headers = {

    "x-rapidapi-key": "your key here",

    "x-rapidapi-host": "jsearch.p.rapidapi.com"

}

# Make the request to the API

response = requests.get(url, headers=headers, params=querystring)

# Check if the request was successful (status code 200)

if response.status\_code == 200:

    # Parse the JSON response

    data = response.json()

    # Pretty print the JSON response to inspect the structure

    print(json.dumps(data, indent=4))  # Inspect the structure of the response

    # Save the JSON response to a file (optional, for later review)

    with open("cybersecurity\_jobs.json", "w") as json\_file:

        json.dump(data, json\_file, indent=4)

    # Open or create a CSV file to write job data

    with open("cybersecurity\_jobs.csv", mode="w", newline="", encoding="utf-8") as file:

        writer = csv.writer(file)

        # Write the headers

        writer.writerow(["Job Title", "Company", "Location", "Date Posted", "Job Description", "URL"])

        # Assuming the jobs data is in the 'data' key

        if 'data' in data:

            jobs = data['data']

            # Iterate over the job results

            for job in jobs:

                # Extract the relevant information from each job

                title = job.get('job\_title', 'N/A')

                company = job.get('employer\_name', 'N/A')

                location = job.get('location', 'N/A')  # You need to inspect and adjust this key

                date\_posted = job.get('date\_posted', 'N/A')  # Inspect and adjust this key

                job\_description = job.get('description', 'N/A')  # Inspect and adjust this key

                job\_url = job.get('job\_apply\_link', 'N/A')

                # Write the row to the CSV

                writer.writerow([title, company, location, date\_posted, job\_description, job\_url])

        else:

            print("No job data found in the response.")

else:

    print(f"Failed to retrieve data: {response.status\_code}")

**Next Steps:**

1. **Run the script and inspect the print(json.dumps(data, indent=4)) output**.
2. **Check the actual keys used for fields like location, date\_posted, and description.**
3. **Replace job.get('location', 'N/A') with the correct key you find in the JSON.**

**Once you confirm the structure, the N/A values should be replaced with the correct data from the API response.**

**NOTE**: At this point, I encountered a few errors/mix-ups:

1. The CSV file ended up showing N/A in the location, date posted and job description fields in excel, even though the Json file extracted the information.

1. The headers in excel did not show up all the necessary criteria that was asked for in the assignment

1. Made a mistake editing the code, ended up getting confusing data outputs so had to modify it to re-run the desired information.

**TROUBLESHOOTING:** These are the changes I made to get the final code script:

1. ADJUST HEADERS: Added in the additional header fields in the "writer.writerow" section in the code so it shows exactly what I want in excel file (removed job description after inspecting the Json because it was outputting unfinished/unclear sentences, and missing information in other areas).

1. ITERATE OVER THE JOB RESULTS: Modified to include city, state, country for the location

1. EXTRACT: Skills, responsibilities, experience, qualifications, remote, on site or contactor( previous CSV did not include this)

**UPDATED/FINAL CODE OUTPUT WITH FIXES**

import requests

import pandas as pd

from datetime import datetime

# Step 1: Extract - Function to Extract Data from API

def extract\_data\_from\_api(url, headers, querystring, pages=5):

    all\_jobs = []

    # Loop through the pages

    for page in range(1, pages + 1):

        querystring["page"] = str(page)  # Update the page number

        response = requests.get(url, headers=headers, params=querystring)

        # Check if the request was successful (status code 200)

        if response.status\_code == 200:

            data = response.json()  # Return the parsed JSON response

            if 'data' in data:

                all\_jobs.extend(data['data'])

        else:

            print(f"Failed to retrieve data from page {page}: {response.status\_code}")

    return all\_jobs

# Step 2: Transform - Helper Function to Extract Skills and Qualifications

def extract\_skills\_and\_qualifications(job\_data):

    potential\_keys = ['job\_required\_skills', 'skills', 'certifications', 'qualifications', 'job\_description']

    collected\_skills = set()  # Use a set to avoid duplicates

    for key in potential\_keys:

        content = job\_data.get(key)

        if content:

            if isinstance(content, list):

                collected\_skills.update(content)  # Add items from the list

            elif isinstance(content, dict):

                for sub\_value in content.values():

                    if isinstance(sub\_value, str):

                        collected\_skills.add(sub\_value)

            elif isinstance(content, str):

                # Split content into sentences to extract specific qualifications or skills

                lines = content.split('.')

                for line in lines:

                    if any(keyword in line.lower() for keyword in ["skill", "qualification", "certification"]):

                        collected\_skills.add(line.strip())

    return ', '.join(collected\_skills) if collected\_skills else 'N/A'

# Step 2: Transform - Improved Experience Extraction

def extract\_experience(job\_data):

    experience = job\_data.get('job\_required\_experience', {})

    if isinstance(experience, dict):

        years = experience.get('years', None)

        exp\_description = experience.get('description', None)

        if years and exp\_description:

            return f"{years} years - {exp\_description}"

        elif years:

            return f"{years} years"

        elif exp\_description:

            return exp\_description

    elif isinstance(experience, str):

        return experience

    # Fallback: Check for experience-related information inside job descriptions

    job\_description = job\_data.get('job\_description', '')

    if "experience" in job\_description.lower():

        # Extract lines with 'experience' from the description

        exp\_lines = [line.strip() for line in job\_description.split('\n') if "experience" in line.lower()]

        if exp\_lines:

            return ' '.join(exp\_lines)

    return 'N/A'

# Step 2: Transform - Combine Responsibilities and Duties

def extract\_responsibilities(job\_data):

    responsibilities\_text = job\_data.get('job\_description', 'N/A')

    if isinstance(responsibilities\_text, dict):

        responsibilities\_text = ' '.join([str(v) for v in responsibilities\_text.values()])

    # Look for specific sections related to duties and responsibilities

    duty\_lines = []

    lines = responsibilities\_text.split('\n')

    capture = False

    for line in lines:

        lower\_line = line.lower()

        if "duties" in lower\_line or "responsibilities" in lower\_line:

            capture = True  # Start capturing when we find relevant headers

        if capture:

            if "requirements" in lower\_line:

                break  # Stop capturing when we hit unrelated sections

            duty\_lines.append(line.strip())

    return ' '.join(duty\_lines).strip() if duty\_lines else 'N/A'

# Helper function to check job status

def check\_status(job\_data, key, keywords):

    if job\_data.get(key, False):

        return 'Yes'

    else:

        description = job\_data.get('job\_description', '').lower()

        return 'Yes' if any(keyword in description for keyword in keywords) else 'No'

# Step 2: Transform - Extract On Site, Full Time, Remote, and Contractor Status

def extract\_job\_status(job\_data):

    full\_time = check\_status(job\_data, 'job\_is\_full\_time', ['full-time'])

    remote = check\_status(job\_data, 'job\_is\_remote', ['remote'])

    contractor = check\_status(job\_data, 'job\_is\_contract', ['contractor'])

    on\_site = check\_status(job\_data, 'job\_is\_on\_site', ['on-site'])

    return full\_time, remote, contractor, on\_site

# Step 2: Transform - Processing the Extracted Data

def transform\_job\_data(jobs):

    job\_list = []

    for job in jobs:

        title = job.get('job\_title', 'N/A')

        company = job.get('employer\_name', 'N/A')

        city = job.get('job\_city', 'N/A')

        state = job.get('job\_state', 'N/A')

        country = job.get('job\_country', 'N/A')

        timestamp = job.get('job\_posted\_at\_timestamp', None)

        date\_posted = datetime.utcfromtimestamp(timestamp).strftime('%Y-%m-%d') if timestamp else 'N/A'

        # Extract Skills and Qualifications (enhanced details)

        skills\_and\_qualifications = extract\_skills\_and\_qualifications(job)

        # Responsibilities and Duties combined under Responsibilities

        responsibilities = extract\_responsibilities(job)

        # Experience (with improved handling of nested information)

        experience = extract\_experience(job)

        # Full Time / Remote / Contractor / On Site status

        full\_time, remote, contractor, on\_site = extract\_job\_status(job)

        # Extract the job application link

        job\_url = job.get('job\_apply\_link', 'N/A')

        # Create a dictionary for each job

        job\_list.append({

            "Job Title": title,

            "Company": company,

            "City": city,

            "State": state,

            "Country": country,

            "Date Posted": date\_posted,

            "Skills and Qualifications": skills\_and\_qualifications,

            "Responsibilities": responsibilities,

            "Experience": experience,

            "Full Time": full\_time,

            "Remote": remote,

            "Contractor": contractor,

            "On Site": on\_site,

            "URL": job\_url

        })

    return job\_list

# Step 3: Load - Writing Processed Data to CSV using Pandas

def load\_data\_to\_csv(job\_list, filepath):

    # Convert the list of job dictionaries into a DataFrame

    df = pd.DataFrame(job\_list)

    # Export the DataFrame to a CSV file

    df.to\_csv(filepath, index=False)

# Main function to execute the ETL process

def main():

    # API request setup

    url = "<https://jsearch.p.rapidapi.com/search>"

    querystring = {

        "query": "cybersecurity jobs on glassdoor",

        "num\_pages": "1",

        "date\_posted": "all"

    }

    headers = {

        "x-rapidapi-key": "your key here",

        "x-rapidapi-host": "jsearch.p.rapidapi.com"

    }

    # Extract: Get job data from API for 5 pages

    raw\_jobs = extract\_data\_from\_api(url, headers, querystring, pages=5)

    if raw\_jobs:

        # Transform: Process the extracted data

        transformed\_jobs = transform\_job\_data(raw\_jobs)

        # Load: Save the processed data to CSV

        load\_data\_to\_csv(transformed\_jobs, "C:/Users/garne/Documents/cybersecurity\_jobs\_etl\_pandas.csv")

    else:

        print("No job data found to process.")

# Run the main function

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Comparison and Breakdown of Updates Made to the Code**

The updated code introduces several enhancements over the original script. Below is a simple breakdown of the changes and updates made.

**1. Modularization and Code Structure**

**Original Code:**

* The script is a linear flow with code executing sequentially.
* Uses a helper function extract\_skills but handles other data extraction inline.

**Updated Code:**

* The code is refactored into modular functions representing each step of the ETL (Extract, Transform, Load) process:
  + extract\_data\_from\_api: Handles data extraction from the API.
  + transform\_job\_data: Processes and transforms the extracted data.
  + load\_data\_to\_csv: Loads the transformed data into a CSV file.
* Additional helper functions for data transformation:
  + extract\_skills\_and\_qualifications
  + extract\_experience
  + extract\_responsibilities
  + extract\_job\_status
* **Benefit:** Improves readability, maintainability, and reusability of the code. Each function has a single responsibility, making debugging and testing easier.

**2. Pagination Handling**

**Original Code:**

* Fetches data from only one page (num\_pages is set to "1").
* Does not handle pagination; limits the dataset to the first page of results.

**Updated Code:**

* Introduces a loop in the extract\_data\_from\_api function to handle multiple pages.
* Accepts a pages parameter (default is 5) to specify how many pages to fetch.
* Updates the querystring["page"] parameter within the loop to fetch subsequent pages.
* **Benefit:** Retrieves a larger dataset by fetching multiple pages, providing more comprehensive data for analysis.

**3. Enhanced Data Extraction and Transformation**

**a. Skills and Qualifications Extraction**

**Original Code:**

* Uses the extract\_skills function to collect skills from predefined keys.
* Handles lists and basic string processing but may collect duplicates.

**Updated Code:**

* Uses extract\_skills\_and\_qualifications function:
  + Collects skills and qualifications from multiple potential keys.
  + Uses a set to avoid duplicate entries.
  + Parses strings to extract sentences containing keywords like "skill," "qualification," or "certification."
* **Benefit:** Provides a more thorough and cleaner collection of skills and qualifications, reducing redundancy.

**b. Experience Extraction**

**Original Code:**

* Extracts experience from job\_required\_experience but may not handle nested structures well.
* Defaults to 'N/A' if experience is not found.

**Updated Code:**

* Uses extract\_experience function:
  + Handles both dictionaries and strings in job\_required\_experience.
  + Checks for 'years' and 'description' to provide detailed experience information.
  + Falls back to parsing the job description if necessary.
* **Benefit:** Offers a more robust extraction of experience requirements, ensuring important details are not missed.

**c. Responsibilities Extraction**

**Original Code:**

* Assigns the entire job description to the responsibilities field without specific parsing.

**Updated Code:**

* Uses extract\_responsibilities function:
  + Parses the job description to extract sections specifically related to duties and responsibilities.
  + Captures text between headers like "Duties" or "Responsibilities" and stops at "Requirements" or similar sections.
* **Benefit:** Isolates relevant information, making the responsibilities field more precise and useful.

**d. Job Status Fields Extraction**

**Original Code:**

* Checks for on\_site, remote, and contractor statuses directly from API fields.
* Does not check the job description for these statuses.

**Updated Code:**

* Uses extract\_job\_status function with a helper function check\_status:
  + Checks both API fields and job description text for keywords.
  + Keywords include variations like "full-time," "remote," "contractor," and "on-site."
* **Benefit:** Improves accuracy in determining job statuses by considering additional information from the job description.

**4. Use of Pandas for Data Loading**

**Original Code:**

* Uses the csv module to write data row by row into a CSV file.
* Manually handles CSV file operations and writes headers.

**Updated Code:**

* Uses Pandas library to create a DataFrame from the list of job dictionaries.
* Writes the DataFrame to a CSV file using df.to\_csv().
* **Benefit:** Simplifies data handling and leverages Pandas' powerful data manipulation capabilities. Makes it easier to perform additional data analysis if needed.

**5. Improved Date Formatting**

**Original Code:**

* Uses datetime.utcfromtimestamp to format the date.
* May not handle missing or incorrect timestamps gracefully.

**Updated Code:**

* Includes a check for the presence of timestamp before formatting.
* Defaults to 'N/A' if the timestamp is missing.
* **Benefit:** Prevents errors related to missing timestamps and ensures the date is consistently formatted.

**6. Additional Improvements**

**a. Error Handling and Logging**

* **Updated Code:** Prints error messages if data retrieval fails for any page, aiding in debugging.

**b. Code Readability**

* **Updated Code:** Includes comments and docstrings that explain the purpose of functions and code blocks.
* Uses more descriptive variable names for clarity.

# **Resources/Research**

**VS Code (For creating/running the scripts)**

**CoPilot**

**Claude**

**ChatGPT**

**YouTube (Visual learning for navigating VS code)**

[**Glassdoor | Job Search & Career Community**](https://www.glassdoor.com/index.htm)

[**Welcome to Rapid’s new Experience! 🚀 (rapidapi.com)**](https://rapidapi.com/hub)

[**GitHub**](https://github.com/)