Web Connector: A Unified API Wrapper to Simplify Web Data Collection

Weiyuan Wu, Simon Fraser University, youngw@sfu.ca
Pei Wang, Simon Fraser University, peiw@sfu.ca
Yi Xie, Simon Fraser University, yi_xie_2@sfu.ca
Yejia Liu, Simon Fraser University, yejia_liu@sfu.ca
George Chow, Simon Fraser University, george_chow@sfu.ca
Jiannan Wang, Simon Fraser University, jnwang@sfu.ca

ABSTRACT

Collecting structured data from Web APIs, such as the Twitter API, Yelp Fusion API, Spotify API, and DBLP API, is a common task in the data science lifecycle, but it requires advanced programming skills for data scientists. To simplify web data collection and lower the barrier to entry, API wrappers have been developed to wrap API calls into easy-to-use functions. However, existing API wrappers are not standardized, which means that users must download and maintain multiple API wrappers and learn how to use each of them, while developers must spend considerable time creating an API wrapper for any new website. In this demo, we present the Web Connector, which unifies API wrappers to overcome these limitations. First, the Web Connector has an easy-to-use programming interface, designed to provide a user experience similar to that of reading data from relational databases. Second, the Web Connector's novel system architecture requires minimal effort to fetch data for end-users with an existing API description file. Third, the Web Connector includes a semi-automatic API description file generator that leverages the concept of *generation by example* to create new API wrappers without writing code.

PVLDB Reference Format:
Weiyuan Wu, Pei Wang, Yi Xie, Yejia Liu, George Chow, and Jiannan Wang. Web Connector: A Unified API Wrapper to Simplify Web Data Collection. PVLDB, 16(12): 4042-4045, 2023.
doi:10.14778/3611540.3611616

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sfu-db/APIConnectors.

1 INTRODUCTION

Data is the primary fuel for any data analysis, making it the most critical and fundamental component of the data science lifecycle. Collecting data from the web is a vital skill for data scientists. Fortunately, there are several websites that provide their data through public APIs [12]. One popular Github repository, public-apis [7], with 231k stars, curates hundreds of free APIs that offer valuable data sources across different domains, such as finance, business, and food & drink.

Although convenient, fetching data successfully from a Web API requires advanced programming skills [13]. For instance, a data scientist who intends to gather all the SIGMOD publications utilizing the DBLP Search API [3] and store them in a Pandas DataFrame must have a comprehensive understanding of various concepts:

(1) `What is the endpoint?` (*https://dblp.org/search/publ/api*)
(2) `What is the request parameter?` (*q=sigmod*)
(3) `How to retrieve more rows using pagination API?` For example, the API request "*https://dblp.org/search/publ/api?q=sigmod&h=100&f=300*" can return 100 publications starting from the 300th publication.
(4) `How to establish a HTTP connection, send requests, and get responses?`
(5) `How to parse a JSON/XML response?`
(6) `How to send requests in parallel to save time?`
(7) `How to handle errors and exceptions? (e.g., what if a request gets rejected or returns an error?)`

There exist two distinct solutions to the challenge of web data collection. One solution involves the adoption of graphical user interface (GUI) applications such as Postman [6], Tableau Web Data Connector [8], and ScrAPIr [11]. These applications facilitate data acquisition from web APIs by non-programmers but are generally unsuitable for data scientists who primarily operate within a Python or R programming environment. Usage of these applications entails manual transfer of data between the GUI application and the programming environment, leading to errors and increased labour demands. Additionally, it introduces difficulties in data provenance and reproducibility of results.

The other solution involves the utilization of an API wrapper, which essentially encapsulates API calls into simple and user-friendly functions(e.g., tweepy for Twitter API [9]). This approach simplifies the process of web data collection for data scientists. Nonetheless, there exists a steep learning curve for data scientists to become proficient in using an API wrapper for each specific website [14]. Furthermore, the availability of wrapper libraries is limited to a handful of popular websites, and collecting data from websites without wrapper libraries remains an arduous task.

In this demo paper, we introduce a novel solution, Web Connector, which serves as a unified API wrapper in Python[1] to address the shortcomings of current solutions for web data collection. Our approach is underpinned by two key observations. Firstly, although web APIs can be highly complex, we only need to handle a specific type of API that *retrieves structured data from a website database*. Through an extensive survey [10] of public web APIs, we have identified a common pattern in relation to query parameters, pagination, and authorization for this type of API. Secondly, several repetitive tasks are common to all API wrappers, including error handling and parallel request management. By implementing these tasks once, they can be reused for multiple API wrappers.

The Web Connector comprises three essential components. Firstly, the execution logic, which contains the code for web data collection and is published as a library. Secondly, the API description files, which describe how to use the execution logic in component 1 to

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611616

[1]Our focus on Python is based on its prevalence in the data science community; however, it is worth noting that our system design can be extended to other programming languages such as R.

download data for a specific API endpoint. Thirdly, an API description file generator that creates API description files for component 2 from examples. The Web Connector boasts several features, including its ease of use through a unified programming interface, its ease of extension through declarative API description files and the aid of the generator, and its ease of update by automatically updating the API descriptions from a registry without needing to update the library. Our extensive survey [10] of commonly used websites, including the 50 most frequently visited websites like Youtube, and 50 randomly sampled domain-specific APIs from a popular Github API hub page, shows that the Web Connector can cover the majority of the website (81 out of 100).

Our library is also presented in PyData Global and the related video can be accessed through [1].

We describe our demo in Section 2 and provide detailed information about our system in Section 3. Finally, in Section 4, we conclude the paper.

2 USING WEB CONNECTOR

In this section, we first introduce how to use Web Connector. After that, we describe our demo, which showcases to the audience how to do data collection using Web Connector.

2.1 A Unified Programming Interface

When a data scientist is given a relational database, it is easy for her to read data from each table in the database. The reason is that the interface to access a relational database is unified. We want to give data scientists the same experience as reading data from a (hidden) database on the website.

Our unified programming interface consists of five *declarative* functions: `connect`, `show_tables`, `show_schema`, `info`, and `query`. That is, a data scientist can use them to describe what she wants to do rather than how to do them.

Step 1. Connect. A data scientist first specifies which website she is interested in by calling the `connect` function.

```
connect(website_name, _auth, _concurrency)
```

The function returns a `Connector` object, which will be used to understand and query the Website database. The function can take as input two optional arguments.

- **_auth** aims to unify authorization. It supports four commonly used authorization schemes. Please refer to Section 3.2 for more detail.
- **_concurrency** aims to simplify concurrency. This field describes how many requests at max are active (request sent but response no received) at *any* time.

Step 2. Understand. The data scientist can check what tables are available in the database by calling the `show_tables` function and what is the schema of each table by calling the `show_schema` function. Additionally, she can check what query parameters can be used to query each table by calling the `info` function.

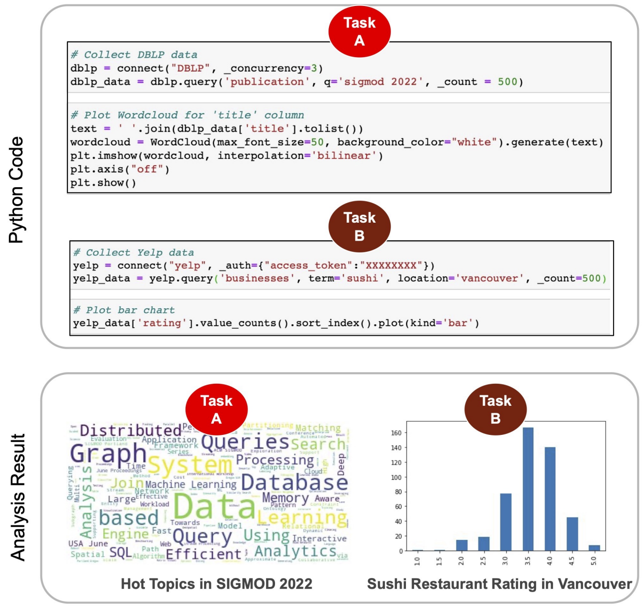
Step 3. Query. Once a table is selected, the data scientist can read data from the table by calling the query function:

```
query(table_name, query_parameter_list, _count)
```

The function unifies query parameters and pagination, respectively.

- **query_parameter_list** is a list of key-value pairs. The query parameters can be a direct mapping of parameters in the request, or a transformation. For the latter, for example, DBLP provides an API to fetch publications of a specific author, e.g., "*https://dblp.org/search/publ/api?q=author:Michael_Stonebraker:*". The two parameters "*first_name = ...&last_name=...*" are merged into a single one "q=...". The query function hides this complexity from the end user.
- **_count**. The user can directly specify the number of returned results without getting into unnecessary detail about a specific pagination scheme. The system will try to send out requests concurrently to make the result meet the requested count.

After collecting all API responses, the query function will automatically extract structured data from JSON/XML responses and return a Pandas DataFrame.



```
# Collect DBLP data
dblp = connect("DBLP", _concurrency=3)
dblp_data = dblp.query('publication', q='sigmod 2022', _count = 500)
```

```
# Plot Wordcloud for 'title' column
text = ' '.join(dblp_data['title'].tolist())
wordcloud = WordCloud(max_font_size=50, background_color="white").generate(text)
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
```

```
# Collect Yelp data
yelp = connect("yelp", _auth={"access_token":"XXXXXXXX"})
yelp_data = yelp.query('businesses', term='sushi', location='vancouver', _count=500)
```

```
# Plot bar chart
yelp_data['rating'].value_counts().sort_index().plot(kind='bar')
```

Figure 1: Case Study on DBLP API and Yelp API

2.2 Demonstration

For the demonstration, we have compiled a set of 119 questions [2], spanning 18 distinct categories, enabling the audience to peruse and experiment with the tool. Each question within the list consists of the question, for instance, "How to fetch all publications of SIGMOD 2022?", accompanied by a code snippet that employs *Connector* to produce the desired response. Throughout the demonstration, we will present these questions to our audience, ask them to select any questions that pique their interest and run the code to get the answer.

We employ the subsequent two tasks to exemplify the process:

A. Collect SIGMOD 2022 paper through the DBLP search API and generate a word cloud from the paper titles.
B. Collect the ratings of the sushi restaurants in Vancouver through the Yelp search API and plot the distribution.

The Python code to perform the tasks described above on Jupyter notebooks is presented in Figure 1.

To execute Task A, the user first invokes the `connect` function to retrieve the DBLP API descriptions and sets `_concurrency` to 3 to ensure a maximum of 3 requests per second. Subsequently, the `query` function can be called with q="sigmod 2022" and `_count`=500. By specifying `_count`, the user can retrieve more than the default 30 results returned by the DBLP API without worrying about pagination. The query result is returned as a Pandas DataFrame, which can be directly used for analysis. Our analysis shows that the hot topics at SIGMOD 2022 include "System," "Graph," "Learning," and "Distributed."

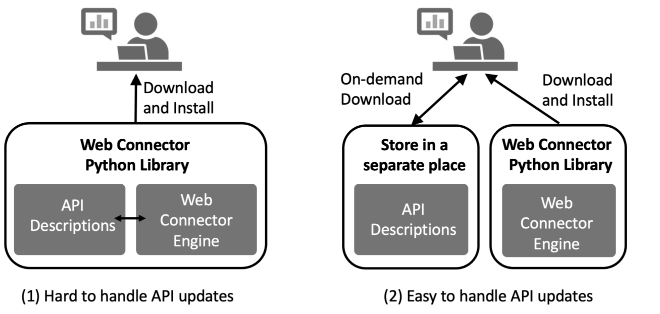


Figure 2: An illustration of two system architecture designs.

To perform Task B, the user can collect data from the Yelp API in a similar manner. Our analysis indicates that most of the restaurants have a rating of 3.5 or 4.

Overall, this demo showcases three advantages of using Web Connector:

(1) The process of collecting data from diverse sites is highly streamlined and unified, requiring only two lines of code to retrieve a Pandas DataFrame from either DBLP or Yelp.

(2) The API call process is exceptionally smooth and user-friendly, featuring automated pagination, error handling, and parallelism. These automated handling mechanisms make the unified interface possible, and fetching data from website APIs is as straightforward as reading data from a relational database.

(3) The organization of results in a Pandas DataFrame and integration with the Python ecosystem make it incredibly convenient to perform downstream data analysis. Users can effortlessly use the output of API requests for further analysis and modeling, which is facilitated by the popular and powerful Python data analysis ecosystem.

3 THE WEB CONNECTOR SYSTEM

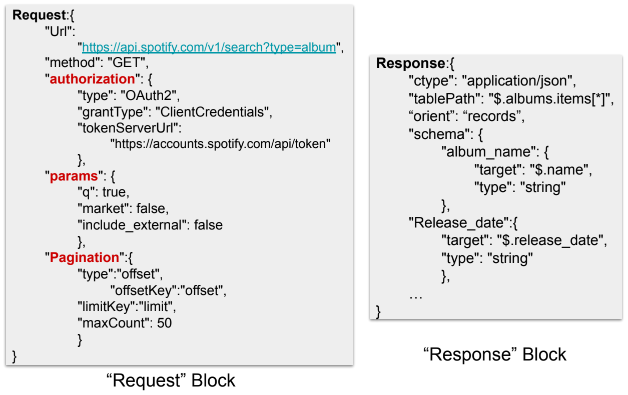
In this section, we first discuss the architecture design of ConnectorX, and then describe the detailed system implementation.

3.1 System Architecture

At a high level, Web Connector consists of two parts. The *API-specific* part covers the specification of each supported API, including the endpoint, the query parameter format, the response data format, the pagination scheme, and the authorization scheme of an API. All these specifications are described in an *API Description*. The *general-component* part covers reusable components that benefit to many APIs, including rate limit handling, error handling, and the implementations of common pagination and authorization schemes. These components constitute the *Web Connector Engine*.

Figure 2 illustrates two architecture design choices. The first design is shown on the left side of the figure. It puts both parts in a single Python library. The main issue of this design is that it is hard to handle API updates. If the specification of a Web API has some changes, the corresponding API description needs to be updated accordingly. Since API descriptions store inside the library, updating them will lead to a new version of the library. As a result, we have to ask all the current users to upgrade to this new version, which is almost impossible.

To address this issue, Web Connector adopts the design shown on the right side of Figure 2. It separates the storage of API descriptions from the Web Connector engine. With this design, the user does not need to upgrade the library. She only needs to delete the local API description that stops working and then call the `connect` function,



```
Request:{
        "Url":
                "https://api.spotify.com/v1/search?type=album",
        "method": "GET",
        "authorization": {
                "type": "OAuth2",
                "grantType": "ClientCredentials",
                "tokenServerUrl":
                        "https://accounts.spotify.com/api/token"
                },
        "params": {
                "q": true,
                "market": false,
                "include_external": false
                },
        "Pagination":{
                "type":"offset",
                        "offsetKey":"offset",
                "limitKey":"limit",
                "maxCount": 50
                }
}
```

“Request” Block

```
Response:{
        "ctype": "application/json",
        "tablePath": "$.albums.items[*]",
        “orient”: “records”,
        "schema": {
                "album_name": {
                        "target": "$.name",
                        "type": "string"
                },
        "Release_date":{
                "target": "$.release_date",
                "type": "string"
                },
        …
}
```

“Response” Block

Figure 3: The API description of Spotify Album Search API.

which will automatically download the up-to-date API description of the API.

3.2 API Description

There can be multiple APIs for one website. We organize the API descriptions for one website under one folder. For the DBLP example, we may have a folder, containing a meta-data file and three API descriptions for the three APIs: publication API, author API, and venue API [3]. The meta-data file stores the meta information of the API descriptions, such as the number and names of the API descriptions. Those API descriptions are stored in a GitHub repository where developers can freely access and download the existing API descriptions. To use the API descriptions, at runtime, the Web Connector engine handles the API description processing, requesting and receiving data.

Figure 3 shows an example of Spotify Album Search API. There are two major blocks: “request” and “response”. “request” block defines the parameters needed for a successful API call. “response” block defines how to parse and flatten the returned nested-structured result to a table format. While the “url” and “method” fields are straightforward, we delve into the details of other fields.

Authorization There is a variety of authorization schemes [15]. In the Figure 3 example, the Spotify album search endpoint adopts OAuth 2.0 client credentials authorization. In the case of DBLP, the authorization field is absent (i.e., no auth) since it does not require authorization.

Pagination Our library supports two mainstream pagination schemes. (1) *Offset-based*: attaching limit and offset in each request. It has two flavours. (1.1) *Offset & Limit*: requesting `limit` the items starting from a specific offset. for example offset = 0, limit = 20 returns the first 20 items. (1.2) *Page & Perpage*: requesting the content of a certain page. (2) *Cursor-based*: The response contains a token to be used for getting more data. (2.1) *Page Cursor*: a token for next “page” is included in the metadata of each response. (2.2) *Item Cursor*: a token for the first item of the next “page” is included in the metadata of each response.

Query Parameters The field “params” defines legal query parameters like search keywords that can be queried through the `query` method. In Figure 3, “q” is the keyword search query parameter, “market” defines that only the content playable in that market is returned, “include_external” include any relevant audio content that is hosted externally. These parameters are usually listed and explained in the API documentation.

Response The response block defines how to parse and flatten the returned nested-structured result to a table format. We support both JSON and XML. Figure 3 shows an example of how to map

JSON data to a table. The "ctype" field says it is a JSON string to be mapped. The "tablePath" field uses a JSONPath expression [5] to specify the location of the table data in the JSON data. Finally, the "schema" field specifies the location of each table attribute in the JSON data and their data types.

3.3 Web Connector Engine

The Web Connector engine is the component that handles API request creation, response parsing, and DataFrame construction. It also contains complex logic for concurrency and error handling.

Concurrent Requesting. A naive approach for request sending is one at a time. In the case of pagination, multiple requests are sent out sequentially. This is slow because it misses the opportunity of sending concurrent requests to the API server. For example, to fetch 1000 records for the keyword "restaurant" using the Yelp Search API, the basic workflow took 32.7 seconds, while by allowing 5 requests per second concurrency, Web Connector finished the task in 6.87 seconds, which is 4.8x faster.

Web Connector makes use of the Python async/await feature, which is the solution to the IO-bounded applications. Web Connector delays all IO operations into an async object and then uses `asyncio.gather` to let the delayed IO operations be executed concurrently.

For the offset-based pagination, Web Connector first creates delayed requests for every offset, and then sends them out using `asyncio.gather` altogether. For the cursor-based pagination, since the next page depends on the last record of the previous response, it is not possible to make it concurrent.

Early Stopping. With pagination on, it is very likely to concurrently send out a large number of requests due to the end of pagination being unknown. This creates resource waste. Web Connector leverages that when there's no more record to return for a certain offset, the response returned is usually different than usual to early stop the pending requests. In detail, Web Connector won't arbitrarily send out requests concurrently, but will only keep a window of `requests` whose paginations are continuous in-flight. This way, Web Connector can early stop as many requests as possible but still maintains the concurrency.

Rate Limit Handling. API endpoints usually post a request-per-second (RPS) restriction on the client. This makes the Web Connector engine at risk of violating the rate limit restriction.

Implementing a request engine that respects the rate limit restriction needs caution. This is because, applying a client-side RPS, i.e. issue at most k requests per second does not necessarily mean that the server will see at most k requests within any given second.

In order to satisfy the server-side RPS, the Web Connector engine uses the window idea in Early Stopping. Web Connector keeps track of the number of *in-flight requests*, denoted by a, and maintains the a to be exactly k until there is no more request to send.

Error Handling. If a rate limit is implicitly enforced, commonly an API server will reject requests with the HTTP 429 error.

Web Connector leverages the exponential backoff idea [4] to tackle this issue. Once any active request receives the 429 error, Web Connector will decrease the current `_concurrency` setting with exponential backoff.

3.4 API Description Generator

The process of writing an API description manually can be tedious and error-prone. There are a number of pain points. Consider the example in Figure 3. The first pain point is that in the "request" block, the developer has to manually construct an API request in a text editor. Second, in the "response" block, the developer has to manually define JSONPath expressions. Third, during testing, the API description may not work as expected.

In response, we propose the idea of *generation by example*, which automatically generates an API description based on input API request examples. The system UI, implemented as a Jupyter widget. Users can generate an API description by first constructing an API request example through filling in the blanks in the step-wise user interface and then clicking the "Send Request" button to send the request to the API server. If a response to the request is successfully returned, the system will automatically generate an API description based on the input request example and the corresponding output response.

Our generation-by-example idea can address the three pain points. For the first pain point, the API request is constructed via the UI rather than a text editor. For the second one, JSONPath expressions are automatically generated rather than manually entered. For the third one, an API description will not be generated until it passes a test case (e.g., a successful response is returned).

It is often not enough to use a single example to generate the complete API description. Connector-Gen enables the developer to build an API description progressively by multiple examples. Specifically, the developer can load an existing API description, which is generated from n examples, into the system. Then, she uses the $(n + 1)$-th example to generate a new API description. By merging this new API description with the existing one, she finally obtains an API description generated from $n + 1$ examples.

4 CONCLUSION

In summary, the focus of our demo paper is to showcase the capabilities of Web Connector, a unified interface that simplifies data collection through web APIs from a multitude of websites. Web Connector achieves this by utilizing reusable settings that are stored in API descriptions, with the added benefit of handling parallelism, errors, and pagination automatically using the Web Connector engine. Additionally, we introduce Connector-Gen, a tool that effectively generates and updates API description files by utilizing examples.

REFERENCES

[1] 2023. A Unified API Wrapper to Simplify Web Data Collection| PyData Global 2020. Retrieved March 16, 2023 from hhttps://www.youtube.com/watch?v=56qu-0Ka-dA

[2] 2023. APIConnectors. Retrieved March 16, 2023 from https://github.com/sfu-db/APIConnectors

[3] 2023. DBLP Search API. Retrieved March 16, 2023 from https://dblp.org/faq/13501473.html

[4] 2023. Exponential backoff algorithm. Retrieved March 16, 2023 from https://en.wikipedia.org/wiki/Exponential_backoff

[5] 2023. JSONPath. Retrieved March 16, 2023 from https://github.com/json-path/JsonPath

[6] 2023. Postman. Retrieved March 16, 2023 from https://www.postman.com/

[7] 2023. public-apis. Retrieved March 16, 2023 from https://github.com/public-apis/public-apis

[8] 2023. Tableau Web Data Connector. Retrieved March 16, 2023 from https://help.tableau.com/current/pro/desktop/en-us/examples_web_data_connector.htm

[9] 2023. Tweepy. Retrieved March 16, 2023 from https://www.tweepy.org/

[10] 2023. Web Connector Survey. Retrieved March 16, 2023 from https://github.com/sfu-db/WebConnectorSurvey

[11] Tarfah Alrashed, Jumana Almahmoud, Amy X Zhang, and David R Karger. 2020. ScrAPIr: Making Web Data APIs Accessible to End Users. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems.* 1–12.

[12] Katrin Braunschweig, Julian Eberius, Maik Thiele, and Wolfgang Lehner. 2012. The State of Open Data Limits of Current Open Data Platforms.

[13] César González-Mora, Cristina Barros, Irene Garrigós, Jose Zubcoff, Elena Lloret, and Jose-Norberto Mazón. 2023. Improving open data web API documentation through interactivity and natural language generation. *Computer Standards & Interfaces* 83 (2023), 103657. https://doi.org/10.1016/j.csi.2022.103657

[14] Adam Marcus, Michael S. Bernstein, Osama Badar, David R. Karger, Samuel Madden, and Robert C. Miller. 2011. Processing and visualizing the data in tweets. *SIGMOD Rec.* 40, 4 (2011), 21–27. https://doi.org/10.1145/2094114.2094120

[15] Prabath Siriwardena. 2020. OAuth 2.0 Security. In *Advanced API Security.* Springer, 287–304.