ScrAPIr: Making Web Data APIs Accessible to End Users

Tarfah Alrashed
MIT CSAIL
tarfah@mit.edu

Jumana Almahmoud
MIT CSAIL
jumanam@mit.edu

Amy X. Zhang
MIT CSAIL
axz@mit.edu

David R. Karger
MIT CSAIL
karger@mit.edu

ABSTRACT

Users have long struggled to extract and repurpose data from websites by laboriously copying or *scraping* content from web pages. An alternative is to write scripts that pull data through APIs. This provides a cleaner way to access data than scraping; however, APIs are effortful for programmers and nigh-impossible for non-programmers to use. In this work, we empower users to access APIs without programming. We evolve a schema for *declaratively specifying* how to interact with a data API. We then develop ScrAPIr: a standard *query GUI* that enables users to fetch data through any API for which a specification exists, and a second GUI that lets users *author* and *share* the specification for a given API. From a lab evaluation, we find that even non-programmers can access APIs using ScrAPIr, while programmers can access APIs 3.8 times faster on average using ScrAPIr than using programming.

Author Keywords

Web APIs; API Description Language; Web Scraping

CCS Concepts

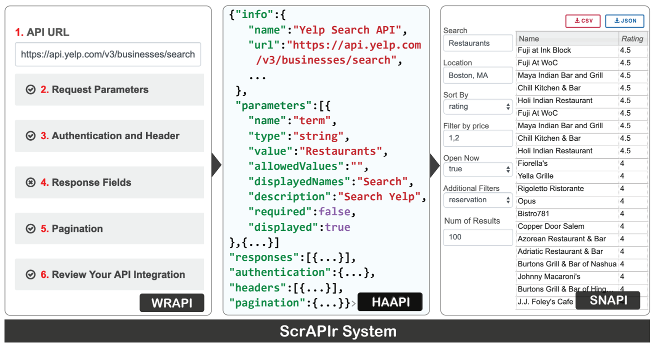
•Human-centered computing → Web-based interaction; •Information systems → RESTful web services;

INTRODUCTION

A common practice on the web is to extract and repurpose structured data from websites. As far back as 2005, the HousingMaps [34] and Chicagocrime [19] *mashups* showed the utility of presenting old data in new visualizations. More recently, applications, articles, and visualizations that repurpose data from sites like Twitter or Wikipedia have become increasingly common [29, 6]. However, a major challenge has been to extract the needed data from its source sites.

Today, the strategies that programmers and non-programmers employ to capture data from websites have diverged. Non-programmers laboriously copy and paste data, or use *web scrapers* that download a site's webpages and parse the content for desired data; many of the early mashups were created this way. But scrapers are error prone as they must cope with ever more complex and dynamic sites and webpage layout changes.

CHI '20, April 25–30, 2020, Honolulu, HI, USA.
© 2020 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6708-0/20/04 ...$15.00.
DOI: https://dx.doi.org/10.1145/3313831.3376691



https://api.yelp.com/v3/businesses/search

```
{"info":{
   "name":"Yelp Search API",
   "url":"https://api.yelp.com
    /v3/businesses/search",
   ...
 },
 "parameters":[{
   "name":"term",
   "type":"string",
   "value":"Restaurants",
   "allowedValues":"",
   "displayedNames":"Search",
   "description":"Search Yelp",
   "required":false,
   "displayed":true
},{...}]
"responses":[{...}],
"authentication":{...},
"headers":[{...}],
"pagination":{...}}
```

Search: Restaurants
Location: Boston, MA
Sort By: rating
Filter by price: 1,2
Open Now: true
Additional Filters: reservation
Num of Results: 100

Name	*Rating*
Fuji at Ink Block	4.5
Fuji At WoC	4.5
Maya Indian Bar and Grill	4.5
Chill Kitchen & Bar	4.5
Holi Indian Restaurant	4.5
Fuji At WoC	4.5
Maya Indian Bar and Grill	4.5
Chill Kitchen & Bar	4.5
Holi Indian Restaurant	4.5
Fiorella's	4
Yella Grille	4
Rigoletto Ristorante	4
Opus	4
Bistro781	4
Copper Door Salem	4
Azorean Restaurant & Bar	4
Adriatic Restaurant & Bar	4
Burtons Grill & Bar of Nashua	4
Johnny Macaroni's	4
Burtons Grill & Bar of Hing...	4
J.J. Foley's Cafe	4

Figure 1. An illustration of the ScrAPIr system, consisting of the components WRAPI, HAAPI, and SNAPI.

While programmers can and do use these non-programmer tactics, they also find ever more sites offering *APIs (Application Programming Interfaces*). Compared to scrapers, APIs provide a more reliable way to access and retrieve clean web data, since they do not rely on parsing complex or irregular web page structures. Many web APIs also offer advanced searches and provide more extensive data than can be accessed by scraping web pages [6].

However, a survey we conducted found three significant obstacles to using APIs: 1) In order to access data through web APIs, people need to write code, making them inaccessible to non-programmers, 2) Even for programmers, web APIs can be hard to learn and use [26, 27] since they vary widely and there is great variability in the quality of the documentation [30, 38, 45], and 3) Most modern web services return structured data in JSON and XML formats, which can contain large and complex hierarchies that are difficult for people to parse without writing code.

To simplify API programming, efforts have been made to design *API description languages* [9, 16] that provide structured descriptions of web APIs. The machine-readable descriptions can be used to generate documentation for programmers [10] and code stubs for connecting to the APIs through various languages. The most popular is the OpenAPI specification [40]. While OpenAPI improves the *developer* experience, it offers no value to non-programmers who still face the necessity of *writing code* to access APIs and parse results.

In this work, we empower users (programmers and non-programmers) to access web APIs without programming. We do so with three related components:

1. We describe how most APIs draw from a small palette of choices: the query parameters used to filter objects, their types, the authentication and pagination methods (if any), and the structure of the returned data. We propose *HAAPI*

(Human Accessible API), an OpenAPI extension for *declaratively specifying* these choices.

2. We build a search GUI called *SNAPI (Search Normalized APIs)* to query and download data from any API with a HAAPI description. SNAPI reads a HAAPI description and presents a typical search form through which a user specifies their query, invokes the relevant API, and shows results in a tabular (spreadsheet) or other standard data format.
3. We build a tool called *WRAPI (Wrapper for APIs)*, that empowers a user to *author* the HAAPI description for an API simply by filling out a separate web form. WRAPI can guide even non-programmers with little or no understanding of APIs to unpack the necessary information from API documentation.

Together, these components form *ScrAPIr*[1], a repository of HAAPI descriptions and GUIs for querying and authoring them. As shown in Figure 1, any user can first use WRAPI to author a HAAPI description of any web API (e.g. the Yelp search API). Once this is done (just once per API), any other user can use SNAPI to query this API and download its response data. ScrAPIr thus empowers users who want data to standardize and access data-query APIs without demanding any cooperation from the sites providing those APIs.

We describe user studies providing evidence that it is easy for users to use the (quite typical) SNAPI search form to retrieve data from HAAPI websites. More significantly, we also find that novice and even non-programmers who are unfamiliar with the concept of an API, using the guidance provided by WRAPI, are able to create HAAPI descriptions for APIs that they had never seen or used before. We also show that programmers can perform the wrapping and query tasks on average 3.8 times faster using ScrAPIr than by writing code.

RELATED WORK

Web Scraping

Early mashup developers wrote one-off web scraping tools, so had to be programmers. Over time, libraries and frameworks for scraping evolved, including Beautiful Soup [37] and Scrapy [39], that continue to support active communities. Other scraping tools are automatic. *Wrapper induction* [21, 2, 20, 44, 1] aims to *train* a web scraper through examples instead of programming. Such tools can infer structure from unstructured or poorly structured text and HTML markup, but their example-based training often leads to errors. Early tools used pattern matching between example content and target content; more recent work [24, 23, 8] leverages *programming by demonstration* to learn not just from the content but also from the steps a human demonstrator takes to extract the content they want. For example, Vegemite extracts tabular data from a web page into a spreadsheet to then allow users to mashup that data with data from other sources [23]. These tools continue to wrestle with the errors inherent in such learned approaches, which can arise from inadequate or inaccurate examples as well as the frequent changes to web content presentations.

[1] http://scrapir.org

Web APIs

As a more robust and accurate alternative for accessing data, many websites now offer APIs that let programmers send a structured query over an HTTP connection and receive structured data in response. Rather than scraping content formatted for humans, programmers face the much easier task of parsing content formatted for machines. However, having to learn and program to a new API for every website is taxing, so efforts have long been underway to *standardize* web APIs. REST offers a web standard for *serializing* API function calls and parameters but standardizes nothing about what those calls and parameters are.

The most aggressive standardization would be for all sites to use exactly the same data API. An ancient standard in this space is SQL. More recently, the Semantic Web [4] proposed a single graph structured data model—RDF—and a standard query language—SPARQL—that would work on any data repository. But the Semantic Web has faced adoption challenges. Even more recently, GraphQL [13] has been put forward; it is too early to know how widely it will be adopted.

Besides simplifying programmer work, a single standard API would also empower non-programmers, who could use a single generic query tool like SNAPI to access data from any compliant site. Although different sites would still use different property names in their data, these could be read out of the site and used to construct site-specific query interfaces. There are a number of search engine interfaces that allow users to query over or explore generic structured data on the web, using GraphQL endpoints [15], the Semantic Web's SPARQL endpoints [12, 18], or DBPedia [3]. However, these tools are limited to sites that expose the single standard API.

Despite these clear benefits, to date no one uniform standard has become dominant. Most websites still offer their own idiosyncratic APIs. This is not surprising. It takes significant effort to standardize, and it is not clear what benefits would accrue to a website doing so. It seems relatively straightforward for a developer to code to a reasonable API (though our surveys below suggest otherwise), so a developer thinking about other developers may consider it sufficient to provide any API at all. But this leaves *non*-programmers without recourse.

API Description Languages

In the absence of a single standard API, it might be possible at least to create a standard way of *describing* the many APIs that are in use. Early efforts include the Web Service Description Language (WSDL) [9] and Web Application Description Language (WADL) [16], both written in XML. More recently, the OpenAPI specification language, API Blueprint [5], and RESTful API Modeling Language (RAML) [35] have emerged. These API descriptions have been used primarily in two ways: first, to automatically generate API connector code in a variety of languages, and second, to automatically generate documentation describing the API to developers. Such documentation may be easier to read than free-form documentation, since everything is standardized. But while these improve the *developer* experience, they offer no value to non-programmers who still face the barrier of *writing code* to access APIs and parse results.

Swagger [41], Postman [31] and RapidAPI [36] share ScrAPIr's use of a meta-schema to generate an API query form. But their intent is to help developers to debug their API queries. Thus, their meta-schemas omit information—such as which parameters are important and how pagination works—that are essential to create an API query form for less experienced users. They also offer form-based meta-schema editors, but they too are aimed at experienced developers documenting and testing their APIs, unlike WRAPI which can guide even novices to describe an API. RapidAPI [36], like ScrAPIr, offers a repository of APIs that users can access but it does not allow users programmers and novices to query and retrieve data from APIs and it is also not free.

Unlike all this prior work, we consider how standardized description of APIs can improve *non-programatic* data access. We show that a suitable description can be used to empower users to *visually* query APIs, and that users (even those unfamiliar with APIs) can *author* this description themselves. We extend the OpenAPI Specification in order to support graphical query access to APIs.

Visually Querying APIs

Prior work tries to facilitate non-programmer use of web APIs. Marmite [44] and Carpé Data [43] have pre-programmed web API connectors to access specific data sources, but non-programmers cannot create connections to new API sources. D.mix [17] also posits a developer for each API connector: "a smooth process for creating the site-to-service maps is important but somewhat orthogonal to this paper's contributions." Modern tools such as Node-RED [29] have brought the Yahoo Pipes style of visual editor for flow-based programming into widespread use, but continue to rely on developers to provide appropriate manifests to describe APIs.

Gneiss [6] does allow users to query arbitrary APIs. But it expects users to do so by typing an API request URL with parameters; it thus requires each user to learn the API and construct API query strings. Gneiss also does not deal with pagination for additional results, API headers, authentication, or presenting a usable query form. From the paper: Gneiss "limits our target users to people who already have some basic knowledge about web APIs,".

These tools thus complement ScrAPIr; ScrAPIr focuses entirely on simplifying access to data, for provision to Gneiss, d.mix, Yahoo Pipes [33], or other data mashup or processing tools for interaction. More similar is Spinel [7], which helps end-users connect data APIs to mobile applications. Spinel has a form-based query-builder for describing an API. However, the resulting description is used to feed data to existing applications, not to build a query interface, and Spinel's meta-schema for describing APIs is missing even more key elements than OpenAPI (e.g. authentication).

None of this previous work directly tackles the core questions in ours: (1) what is the full set of properties needed in an API meta-schema to automate generation of an easily usable data-query UI, and (2) can end users be guided, and how, to provide all the needed properties? By addressing these two questions, ScrAPIr shows how to refactor the API usage workflow so that one relatively inexperienced "gardener" [28] can tackle the steps needed to let all other users make simple form-based queries.

EXPERIENCES AND STRATEGIES WITH WEB DATA

We began by investigating people's needs for and experiences with retrieving web data by distributing a survey through various private university mailing lists, social media, and to individuals identified as using web data as part of their work (e.g. journalists). The survey was taken by 116 people with a wide range of educational and programming backgrounds (31% have no programming skills, 33% are beginners, and the rest are skilled).

We asked *what* web data subjects wanted and *why*. We then asked *how* respondents tried to access that data and whether they succeeded. For two complex questions, we gathered free responses. Two authors then coded the responses into categories (e.g. requires programming skills). The inter-rater reliability (IRR) was 78% for one question and 87% for the other.

Reasons and Methods for Accessing Web Data

Respondents needed web data for many reasons, including data visualization (38%), analysis purposes (22%), filtering (21%), and creating applications based on the data (16%). Overall, 90% of respondents expressed a desire to access web data but only 43% had actually attempted to do so. This gap is not fully explained by lack of education or technical ability, as 70% of respondents reported at least a beginner level of programming experience. Of those who attempted to access web data, only 6% were always successful, while 22% were never successful. These results imply that while there is strong interest in using web data, there exist barriers to doing so, even among those who have the technical skills to complete the task.

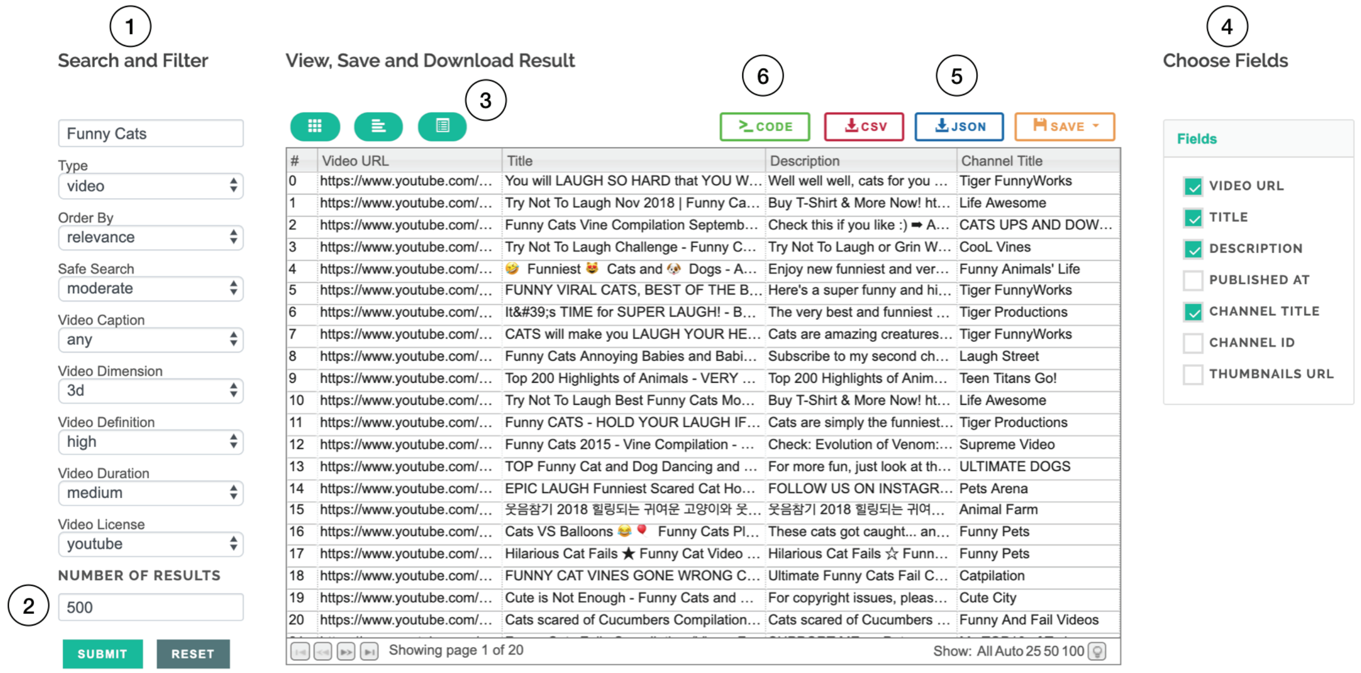
Of the 39 websites from which respondents were able to retrieve data, 22 have APIs (e.g. Twitter, YouTube, Amazon, Wikipedia, etc). Nevertheless, only 27% of our subjects (86% of them programmers) chose to use them. Instead, the majority used alternative methods including scraping (31%), manually copying and pasting (14%), third party websites, plugins or libraries (14%), or a website-provided download option (13%). This suggests that while APIs are often available, they are underutilized.

Obstacles to Accessing Web Data by Scraping

27% of our respondents had tried to scrape one or more websites but were unsuccessful. Some of the main obstacles they reported include: contending with complex HTML structures on the page to get at the data (28%); needing greater programming skills (20%); finding it too time-consuming (20%); finding it brittle when web pages change (5%); and getting blacklisted from scraping (4%). Most of these limitations can be avoided with web APIs, and yet most of our respondents did not use APIs, and among those who tried, many still failed.

Obstacles to Accessing Web Data using APIs

Of the 32% of respondents who previously tried to use APIs, 84% failed. Two of the main obstacles were that dealing with



Funny Cats

Type	video
Order By	relevance
Safe Search	moderate
Video Caption	any
Video Dimension	3d
Video Definition	high
Video Duration	medium
Video License	youtube
NUMBER OF RESULTS	500

#	Video URL	Title	Description	Channel Title
0	https://www.youtube.com/…	You will LAUGH SO HARD that YOU W…	Well well well, cats for you …	Tiger FunnyWorks
1	https://www.youtube.com/…	Try Not To Laugh Nov 2018 | Funny Ca…	Buy T-Shirt & More Now! ht…	Life Awesome
2	https://www.youtube.com/…	Funny Cats Vine Compilation Septemb…	Check this if you like :) ➡ A…	CATS UPS AND DOW…
3	https://www.youtube.com/…	Try Not To Laugh Challenge - Funny C…	Try Not To Laugh or Grin W…	CooL Vines
4	https://www.youtube.com/…	🤣 Funniest 😻 Cats and 🐶 Dogs - A…	Enjoy new funniest and ver…	Funny Animals' Life
5	https://www.youtube.com/…	FUNNY VIRAL CATS, BEST OF THE B…	Here's a super funny and hi…	Tiger FunnyWorks
6	https://www.youtube.com/…	It&#39;s TIME for SUPER LAUGH! - B…	The very best and funniest …	Tiger Productions
7	https://www.youtube.com/…	CATS will make you LAUGH YOUR HE…	Cats are amazing creatures…	Tiger FunnyWorks
8	https://www.youtube.com/…	Funny Cats Annoying Babies and Babi…	Subscribe to my second ch…	Laugh Street
9	https://www.youtube.com/…	Top 200 Highlights of Animals - VERY …	Top 200 Highlights of Anim…	Teen Titans Go!
10	https://www.youtube.com/…	Try Not To Laugh Best Funny Cats Mo…	Buy T-Shirt & More Now! ht…	Life Awesome
11	https://www.youtube.com/…	Funny CATS - HOLD YOUR LAUGH IF…	Cats are simply the funniest…	Tiger Productions
12	https://www.youtube.com/…	Funny Cats 2015 - Vine Compilation - …	Check: Evolution of Venom:…	Supreme Video
13	https://www.youtube.com/…	TOP Funny Cat and Dog Dancing and …	For more fun, just look at th…	ULTIMATE DOGS
14	https://www.youtube.com/…	EPIC LAUGH Funniest Scared Cat Ho…	FOLLOW US ON INSTAGR…	Pets Arena
15	https://www.youtube.com/…	웃음참기 2018 힐링되는 귀여운 고양이와 웃…	웃음참기 2018 힐링되는 귀여…	Animal Farm
16	https://www.youtube.com/…	Cats VS Balloons 😂 🎈 Funny Cats Pl…	These cats got caught... an…	Funny Pets
17	https://www.youtube.com/…	Hilarious Cat Fails ★ Funny Cat Video …	Hilarious Cat Fails ☆ Funn…	Funny Pets
18	https://www.youtube.com/…	FUNNY CAT VINES GONE WRONG C…	Ultimate Funny Cats Fail C…	Catpilation
19	https://www.youtube.com/…	Cute is Not Enough - Funny Cats and …	For copyright issues, pleas…	Cute City
20	https://www.youtube.com/…	Cats scared of Cucumbers Compilation…	Cats scared of Cucumbers …	Funny And Fail Videos

Showing page 1 of 20　　Show: All Auto 25 50 100

Fields
- ✓ VIDEO URL
- ✓ TITLE
- ✓ DESCRIPTION
- PUBLISHED AT
- ✓ CHANNEL TITLE
- CHANNEL ID
- THUMBNAILS URL

Figure 2. The SNAPI UI. (1) Set search parameters. (2) Choose the number of results. (3) View results as table, tree, or list. (4) Choose response fields. (5) Save and download the data in JSON or CSV. (6) Get a code snippet in JavaScript or Python that issues this query.

APIs required advanced programming skills (34%), which many of our respondents did not have, and is time consuming (15%). Both of these obstacles are overcome by ScrAPIr. Another obstacle was the challenge of following the API documentation (9%). We replicated this finding in our system evaluation described later in this paper. This also confirms prior research investigating the learnability and usability of APIs [30, 38]. Other obstacles were related to restrictions that the APIs present, such as rate limits (13%), APIs not returning desired data (10%), authentication requirements (8%), a paywall (7%), getting data in an unusable format (1%), and API changes (1%). And of course, many websites do not provide APIs at all.

THE SCRAPIR ECOSYSTEM

ScrAPIr empowers users to retrieve data from web APIs without programming. It consists of three related components: HAAPI, SNAPI, and WRAPI. HAAPI is a standard ontology for *describing* an API in enough detail to automatically build a graphical interface to query it, without per-API programming. SNAPI is a tool that creates such a GUI from any HAAPI description. And WRAPI is a form-based tool that lets users author HAAPI descriptions of APIs, again without programming. These three components are connected. One uses WRAPI to describe an API. WRAPI then generates a corresponding HAAPI description that can be read by SNAPI. Then, all other users can query that API through SNAPI.

SNAPI

SNAPI is a GUI that can query any API with a HAAPI description. Users select from a list of published APIs to arrive at the page shown in Figure 2 (in this case the YouTube search API). SNAPI provides a typical query interface with typical presentation and functionality described in the caption. Its only difference is that it returns data in a table for easy download instead of as formatted html. We intentionally limit SNAPI to data queries, with no functionality for editing or computing over the data. There are many high quality tools that people can use for these tasks once they have the data.

SNAPI Evaluation

We evaluated SNAPI in a study with 10 participants (5 female, 5 male, ages 18 to 54), with varying programming skills ranging from non-programmer to skilled. We devised five data retrieval tasks that test different functions:

- **Task 1 (search and filter)**: Get a list of the top 300 highly rated Italian restaurants in Boston that are open now at Yelp. Only show the name of the restaurant and the page URL.
- **Task 2 (download data)**: Download a list of 500 Romantic Fiction ebooks from Google Books ordered from the newest to the oldest, in a CSV file.
- **Task 3 (publish queries)**: Search for 25 New York Times news articles about Trump. Only show the title and URL, and save the data publicly.
- **Task 4 (fork and edit queries)**: Search for the dataset that you have just saved, order the articles oldest to newest, and save it under a different name.
- **Task 5 (refine using operators)**: Get the 100 most viewed YouTube videos of Cute pets but not monkeys. Only show the video title and URL.

Before introducing SNAPI, we asked participants how they would perform Tasks 1 and 2. We then demonstrated SNAPI and had each participant perform all 5 tasks.

Results

Before presenting SNAPI, we asked subjects how they would perform Tasks 1 and 2. For Task 1, seven of 10 said that they would go to the Yelp/Google Books website, search for "Italian restaurants"/"Romantic Fiction", then copy and paste each result. For Task 2, they said they would not know how to download the data without a website export button. A few people suggested copy-and-pasting the results in a spreadsheet then saving it. Only two participants indicated that they might check whether the website had an API that they could use, and one suggested using a scraping script available online.

Despite most being unfamiliar with APIs, all participants were able to perform all the tasks using SNAPI with the exception of Task 5 which only 6/10 were able to perform correctly. In Task 5, to search YouTube for "cute pets but not monkeys", participants had to use the special operator "-" to exclude videos: "cute pets -monkeys". This means users had to be aware of special semantics in the query string. This obstacle would also have arisen using YouTube's own search form.

We asked our participants to give us their feedback about SNAPI. One common response was that this tool could be helpful for data visualization, where the first challenge users face is getting the data in a usable format. One participant said "*I have to learn JavaScript and I never had the chance to spend time on that*" A second common response was that it could support users in their workflow process. One said "*This tool would be so helpful for journalists. Some people do this for a living and I would love to use this tool!*" Two stated that SNAPI could serve as an educational step to a better understanding of how APIs work and what kind of data they offer.

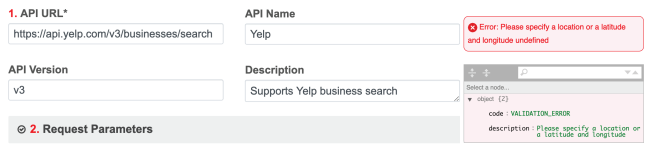
Because SNAPI works just like most web search interfaces, our subjects had little difficulty using it. Rather than a novel UI, the key challenge is determining how to *connect* SNAPI to arbitrary web APIs, as we discuss next.

WRAPI AND HAAPI

Writing code to connect to an API is hard for programmers and impossible for non-programmers. In this section, we describe an alternative. Many APIs are assembled from a tiny palette of possible design choices. We show how to *describe* those design choices in a machine-readable fashion. SNAPI can read that description and use it to generate both a suitable query interface and appropriate API calls. In this section, we simultaneously introduce *HAAPI (Human Access to APIs)*, an ontology for describing an API that SNAPI can query, and *WRAPI (Wrapper for APIs)*, a tool that users can use to provide the necessary information to describe an API.

Describing an API

Our API description ontology extends the *OpenAPI* ontology that emerged recently as part of the Swagger project [40]. The goals of Swagger are twofold. First, it seeks to schematize the information needed for effective *documentation* of an API.



1. API URL*
https://api.yelp.com/v3/businesses/search

API Name
Yelp

API Version
v3

Description
Supports Yelp business search

2. Request Parameters

Error: Please specify a location or a latitude and longitude undefined

```
Select a node...
▼ object {2}
      code : VALIDATION_ERROR
      description : Please specify a location or
                    a latitude and longitude
```

Figure 3. Providing the API URL, and other basic information. When the user enter the URL, WRAPI will display the response to an API query (right), in this case asking the user to provide the required parameter.

Like JavaDoc before it, OpenAPI lets users specify the parameters of a particular API call, their types, and so forth. If a user provides this structured specification, Swagger can automatically generate human-readable documentation of the API for use by developers who want to access it. A second use of this information is for *stub generation*. Given a description, Swagger is able to generate code in a variety of languages that will query the API over the web and return the results in a language-suitable format.

OpenAPI provides much information necessary for SNAPI, but is not sufficient. OpenAPI does not specify information like human-readable parameter names that is necessary for creating a GUI to query the API. And since it is assumed that the code will be integrated into a larger system that uses the API invocations, OpenAPI does not specify information about how to *authenticate* to the API or *paginate* to gather large numbers of results.

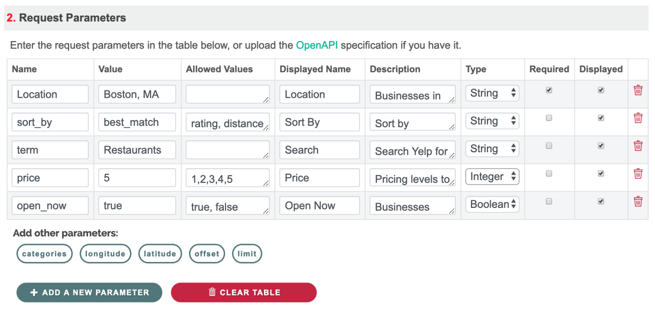
We organize this section by stepping through an API-driven data query and the information needed to carry out each step. In some cases the information is part of OpenAPI; in others we explain how HAAPI extends the schema to include that information. We use `monospace font` to indicate property names in HAAPI. Then, we explain how WRAPI gathers that information from an *integrator* of a given API (who may not be a programmer). In general, the integrator cannot be expected to *know* this information. Instead, we expect that the integrator will seek out and read API documentation in order to find it. WRAPI guides the integrator toward what they should look for at each step of the integration.

API Endpoint

The first bit of information needed to access a web API is its *endpoint*. Any such API is invoked by requesting a suitable URL from the server. OpenAPI uses `host`, `basePath`, and `paths` parameters in its schema to specify the URL to request, such as `https://www.googleapis.com/youtube/v3/search`. WRAPI presents a form where the integrator can enter the API URL, as shown in Figure 3.

Request Parameters

In addition to the URL, the API request contains *query parameters*—arguments to the API method being invoked. These can include, for example, a search string to filter results or a specification of the sort order. Each parameter has a `name` and a `type` such as free text, numeric, or an enumerated list of permitted values. Some parameters are optional and some



2. Request Parameters

Enter the request parameters in the table below, or upload the OpenAPI specification if you have it.

Name	Value	Allowed Values	Displayed Name	Description	Type	Required	Displayed
Location	Boston, MA		Location	Businesses in	String	✓	✓
sort_by	best_match	rating, distance	Sort By	Sort by	String	☐	✓
term	Restaurants		Search	Search Yelp for	String	☐	✓
price	5	1,2,3,4,5	Price	Pricing levels to	Integer	☐	✓
open_now	true	true, false	Open Now	Businesses	Boolean	☐	✓

Add other parameters:

Figure 4. Providing request parameters for Yelp search API in WRAPI. These parameters were retrieved from GitHub. The most used ones in GitHub are added to the table and the rest are added under the table.

required. The OpenAPI schema already represents this information.

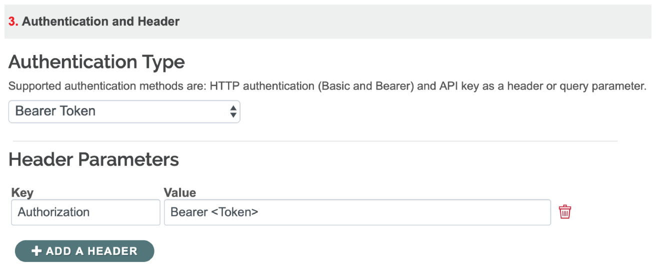
To make a query with these parameters, SNAPI must gather them from the user making the query. SNAPI uses `type` information to specialize parameter entry widgets—for example, providing a dropdown for selecting from an enumerated type. We further extend the OpenAPI schema to include a `displayedName`, a human-readable name property for each parameter, and a default `value`. Both are used in the SNAPI query form. Often, some parameters are esoteric and unlikely to change, so we extend HAAPI to indicate which parameters should be `displayed` in the query UI. Those that are not displayed always take their default values in a query. WRAPI collects this information with a form where users can enter parameter names, default values, and types and indicate which are required, as shown in Figure 4.

Mining GitHub and APIs Guru

Because manually entering parameters can be tedious for the integrator and may require wading through substantial documentation, we offer some alternatives. When the integrator provides the API URL in the first step, WRAPI tries to automatically pull parameter information from two sources: APIs Guru and Github. APIs Guru [14] is a repository of OpenAPI descriptions for thousands of endpoints. If WRAPI finds a description for its endpoint there, it imports the parameter descriptions. At the same time, WRAPI queries Github for code snippets containing the API URL and incorporates parameters it finds.

APIs Guru and Github complement each other. APIs Guru is for documentation; its descriptions are therefore relatively complete. But documentation becomes outdated and may be wrong [27]. Github provides (hopefully) working code examples, so we have higher expectations of correctness. But we will only see parameters that programmers actually use, so the information may be incomplete.

To asses the reliability of APIs Guru we picked a random sample of 10 API descriptions (out of about 1500). All of the 10 OpenAPI specifications listed the right parameters, 2/10 were missing some important information about those parameters (e.g. if the parameter is required). Only 5/5 had the correct authentication type (discussed below, suggesting that information is unreliable. As a result, we collect parameter



3. Authentication and Header

Authentication Type

Supported authentication methods are: HTTP authentication (Basic and Bearer) and API key as a header or query parameter.

Bearer Token

Header Parameters

Key	Value
Authorization	Bearer <Token>

Figure 5. Providing authentication and headers information for Yelp API using WRAPI.

information from APIs Guru, but not authentication information.

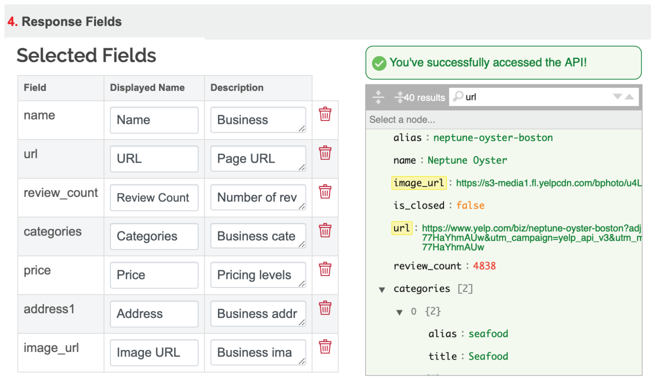
APIs often include many parameters that are optional and unimportant. To address this, WRAPI uses its search of Github to determine the popularity of each parameter. Then, to avoid daunting the user, WRAPI shows only the five most popular parameters in the parameter table, then provides a list of the other parameter names for the integrator to choose and add to the table if they wish. The OpenAPI specification does not include the `displayedName` and `pagination` information (discussed below), but the integrator can add those using WRAPI's form.

Authentication and Headers

Most Web APIs aim to track and control access using some kind of token provided to each API user that is then included with the API request to identify who is making it. There are different types of authentication: HTTP authentication (Basic and Bearer), API key, and OAuth. There are two ways to deliver this authentication: it can be included as one of the named parameters in the query, or it can be provided as a header in the HTTP request. The OpenAPI specification describes the authentication `type`, `name`, and whether it is `query` or `header` parameter. WRAPI uses another form interaction to collect the relevant information, as shown in Figure 5.

To prevent an API from being overwhelmed, API owners often enforce a *rate limit* on the number of requests users can consume. If sites apply rate limits by token, this single token will rapidly be exhausted by multiple users. Thus, we extended OpenAPI to represent instructions for getting an API key (what web page to visit, what to ask for, etc.) WRAPI collects this information from the API integrator. The integrator (who will need to generate a key at least to test their API) can "donate" that key for public use or keep it private. If no public API key is available, SNAPI uses the new HAAPI information to guide a user to get their own (which they too have the option of donating).

We expect authentication to be one of the most significant challenges for an integrator. Unlike the specification of parameters, it does not have an analogue in users' typical interactions with a web search form. The documentation describing authentication can be technical and opaque. And the integrator may have to go through a complex process—signing up on the website, activating a "developer account" and submitting a request—to get an access token. As we show in our user



4. Response Fields

Selected Fields

Field	Displayed Name	Description
name	Name	Business
url	URL	Page URL
review_count	Review Count	Number of rev
categories	Categories	Business cate
price	Price	Pricing levels
address1	Address	Business addr
image_url	Image URL	Business ima

You've successfully accessed the API!

40 results url

Select a node...

```
alias : neptune-oyster-boston
name : Neptune Oyster
image_url : https://s3-media1.fl.yelpcdn.com/bphoto/u4L
is_closed : false
url : https://www.yelp.com/biz/neptune-oyster-boston?adj
      77HaYhmAUw&utm_campaign=yelp_api_v3&utm_m
      77HaYhmAUw
review_count : 4838
▼ categories [2]
  ▼ 0 {2}
      alias : seafood
      title : Seafood
```

Figure 6. Choosing response fields using the interactive pane (right), which shows the API response. The user can search and browse for fields, click on them, then they will be added to the table (left).

studies, non-programmers can succeed in this process, but it is certainly a pain point.

Response

Most web APIs return results in JSON format [22]. As JSON is self describing, it is straightforward for a system to parse those returned results. However, to present those results to a user through SNAPI, some additional information is useful. In particular, many APIs return extremely *rich* JSON, full of properties that are unlikely to interest the user. And, the property names may be relatively opaque. Thus, HAAPI extends the OpenAPI specification with a `responses` property that captures which fields a user is likely to care about and gives them human-readable names and descriptions.

WRAPI provides affordances to simplify extraction of specific result fields. Once the integrator has specified query parameters/authentication, WRAPI submits a sample query and shows the JSON response in an interactive pane, shown in Figure 6 (right), where the user can click to select desired fields. The chosen fields are added to the table on the left, where the user can add `displayed names` and `descriptions` to these fields. WRAPI adds the `name`, `displayed name`, and `description` of these fields to the HAAPI description. SNAPI uses this information to present results by populating the selected result fields and showing them in its UI (Figure 2(4)). This allows a user to further filter the result columns.

Pagination

Most APIs use some form of *pagination* to reduce unnecessary data delivery. Abstractly, there is an ordered sequence of results to a query, and pagination specifies a small subsequence of this ordered sequence to return in the query results. Many web users are familiar with pagination, as it is also used in web search interfaces. Pagination interfaces usually offer “next page” and “previous page” links, and sometimes a way to jump to a specific page number. Pagination is similarly used within APIs. As with authentication, we found only a few distinct pagination mechanisms. Some APIs use a fixed or specifiable number of items per page and a parameter to specify the page number. Others provide next-results and previous-results values that are used to assign the `currentPage` property to request an adjacent subsequence of items. Others have a parameter that specifies either a page `index` or an `offset` into the entire result list along with a `count` of a number of results to return.

Pagination is *not* part of the OpenAPI specification. More precisely, although OpenAPI can specify the parameters that the API happens to use for pagination, it does not provide any way to express that those parameters have pagination semantics. This is understandable; OpenAPI is intended to support arbitrary APIs, while pagination is a semantics specifically directed towards large query result sets. Thus, we extend HAAPI with a `pagination` property that can be populated with the relevant information: the `type` of pagination used, and the `parameter` names used to drive that pagination.

WRAPI provides a form for the integrator to select a particular type of pagination used by the API, then provide the relevant parameter names (Figure 7). If the integrator omits pagination information, SNAPI issues queries but can only retrieve the (default) first page of results. If the integrator provides the pagination information, SNAPI will display a number of results field in its front end (Figure 2(2)) where users can type in the number of results they want to retrieve. SNAPI uses pagination to automatically retrieve as many results as the SNAPI user requests (up to some limit imposed by the API).

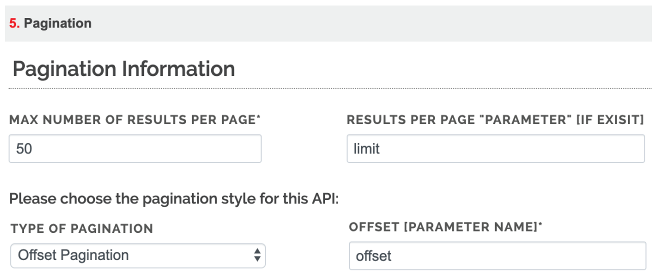
Feedback during Integration

We have described five steps an integrator goes through to generate all the necessary HAAPI information. In an early prototype, similar to other tools like Postman and Swagger, we took the user through all five steps before generating any query to the API. But evaluation of this prototype found that integrators made many integration errors which were difficult to debug. Thus, to provide feedback *during* integration, we added a panel, as shown on the right in Figure 3 and 6, that displays the JSON response to an API query. WRAPI sends an initial query, with no parameters, as soon as the user provides the API url, since even at this early point some APIs begin returning useful responses. As the user provides more information, the query is refreshed. The response is in JSON, so we parse it and display it to the user in a more readable format.

When the API returns an error response, WRAPI looks for a human-readable error message and shows it to the user above the response pane. We propose a simple heuristic to get a human-readable message from the JSON reponse: look for the longest text string. To test this heuristic, we examined the ProgrammableWeb [32], a catalog of over 22 thousand web APIs. Focusing on the *search* category (3534 APIs), we chose a random sample of 40 APIs and found that the longest text heuristic always works but that some APIs return a “success” response with an error message, instead of an “error” response, which is a design limitation of these APIs. Thus, overall our approach works 87.5% of time (with a 95% confidence interval of $\pm$14.78%).

IMPLEMENTATION

ScrAPIr has a front end web interface, built using JavaScript, HTML, and CSS, and a back end component, built using the Firebase Realtime Database, which stores the HAAPI


5. Pagination

Pagination Information

MAX NUMBER OF RESULTS PER PAGE*: 50

RESULTS PER PAGE "PARAMETER" [IF EXISIT]: limit

Please choose the pagination style for this API:

TYPE OF PAGINATION: Offset Pagination

OFFSET [PARAMETER NAME]*: offset

Figure 7. Providing pagination information for Yelp search API using WRAPI.

descriptions. SNAPI reads a HAAPI description from Firebase and uses it to generate the SNAPI front end. SNAPI displays results in a table format (Figure 2(3)), using SlikGrid [25], a JavaScript grid/spreadsheet library. For the interactive pane in Figure 6 (right), we used JSON Editor [11], a tool to view and edit JSON data.

HAAPI EVALUATION

HAAPI should be evaluated by how well it is able to describe web data query APIs. HAAPI builds heavily on OpenAPI, which has already demonstrated broad uptake in the developer community, and makes only limited extensions, primarily to cover pagination, authentication, and presentation. A key metric is coverage: what portion of the APIs provided on the web can be described by HAAPI with sufficient fidelity to enable SNAPI to query them from their descriptions?

To answer this question, we again sampled the ProgrammableWeb and found that 90% (with a 95% confidence interval of ±10.73%) of search APIs can be described by HAAPI such that SNAPI can query them. Of the three not describable by HAAPI, one used a different style of pagination. And two required passing parameters as part of the URL path— `www.example.com/parameter1Value1/parameter2Value2` as opposed to the query string `www.example.com?parameter1=Value1&parameter2=Value2`—which is currently not supported by HAAPI. It would be trivial to extend HAAPI to include path-based parameters, which would improve coverage of the sample to 97% (with a 95% confidence interval of ±6.42%).

WRAPI EVALUATION

We conducted a user study comparing WRAPI to the typical approach of coding to access APIs. We focused on the following questions: (i) Can first-time users who are programmers access and query APIs using WRAPI faster than writing code?, and (ii) Can first-time users without programming skills use WRAPI to access and query web APIs? We recruited 20 participants (9 male, 11 female, ages 21 to 62) for a two-hour user study. Ten of our participants were non-programmers, and ten programmers. Programmer skills ranged from intermediate to advanced: 0 newcomers (learning concepts), 0 beginners (understands concepts but frequently needs documentation), 2 intermediate (does not usually need documentation), 2 skilled (can write complicated programs), and 6 advanced (professional). Programmer experience with APIs varied: 2 had never used APIs, 2 had used them once, and 6 had used them a couple of times or more.

Procedure

Non-programmers only used WRAPI while programmers had one session with WRAPI and another writing code. In the coding session, participants could use any language, online code snippets, or libraries. The same API and task was assigned for both sessions so we could compare within subjects. Participants were assigned one of 3 tasks:

- **Yelp**: Get a list of the top 300 highly rated Italian restaurants in NYC that are open now. Only get the name, URL, and rating for each restaurant.
- **Reddit**: Get a list of the newest 200 posts about GRE exam tips on Reddit. Only get the title, subreddit, URL, and number of subscribers for each post.
- **Google Books**: Get a list of 250 ebooks about Human History from Google Books. Only get the title, published date and the info link for each ebook.

We chose these three APIs because they are different in terms of what they require from the user to access them (e.g., some require authentication and others do not) and their documentation quality.

With our programmer participants, we counterbalanced the order of methods used to perform the tasks: half of our participants started by using WRAPI and the other half by writing code. We assigned each task to participants with different programming skill levels (e.g. we assigned the Google Books task to 3 participants, 1 advanced, 1 skilled, and 1 intermediate). The WRAPI session began with a quick demo of the tool. We limited the time for each session: if a participant did not complete the task with a given method within an hour, we moved on to the next session. After finishing their sessions, each participant answered a survey that asked them to reflect on the methods' usability and efficiency.

Because we expected that the time spent on the first session would be highly affected by the time spent reading and understanding the API documentation, while it would already be known in the second session, we decided to measure the time spent on documentation separate from the time spent using WRAPI and writing code. We accomplished that by using TimeYourWeb [42], a Chrome extension that tracks the time spent on web pages. The majority of documentation time was spent in the first session.

Results

All 20 participants completed their task using WRAPI and 9 of the 10 programmers completed it writing code. Participant P5 did not finish their coding task despite rating their programming as skilled. Figure 8 shows the time spent using WRAPI and (for programmers) writing code, and the time they spent on the API documentation. The top shows averages by task/API while the bottom shows individual times.

All programmers (whether they started with WRAPI or coding) spent more time coding than reading the documentation, but

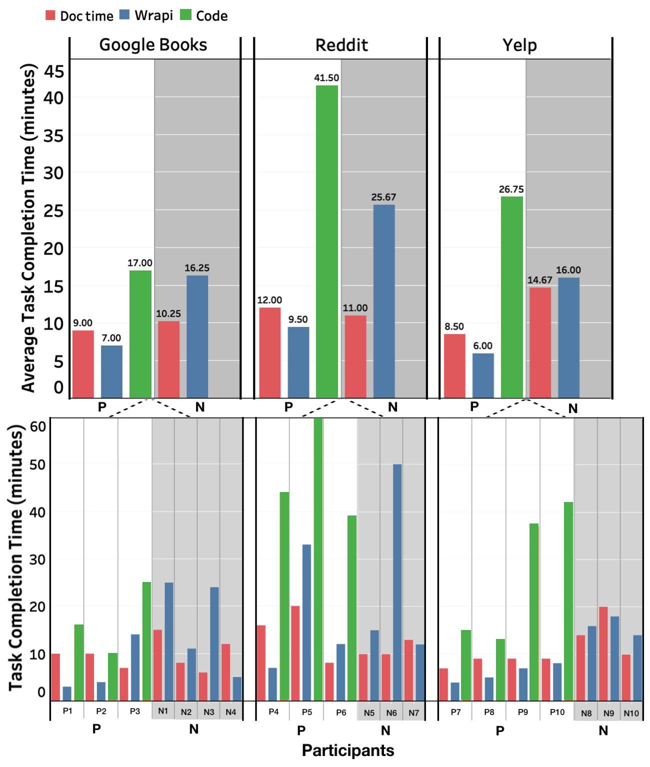


Figure 8. The top chart shows the average time programmers (P) and non-programmers (N) spent on completing their task using WRAPI, writing code (for programmers), and reading the API documentation. The bottom chart shows in detail the time each participant spent.

less time using WRAPI than reading documentation. For the programmers who finished both the WRAPI and coding tasks, the average time they spent to complete their task using WRAPI was 7.1 minutes versus 26.8 minutes by coding, a factor of 3.8x faster. Programmers completed the WRAPI task 2.4 times faster than coding for Google Books, 4.4 times faster for Reddit, and 4.5 times faster for Yelp. We conducted a paired t-test for programmer participants who completed both tasks. There was a significant difference in the completion times for writing code (M=26.778, SD=13.746) and using ScrAPIr (M=7.111, SD=3.756) with p=0.001.

Programmers were generally faster than non-programmers using WRAPI. But the average time non-programmers spent to complete their (WRAPI) task was 19 minutes—faster than programmers writing code.

In the survey, we asked participants to rate how usable they found WRAPI. Participants answered all questions with a five-point Likert scale, with 1 indicating the tool was very easy to use and 5 very difficult to use. The average ratings were 1.7 and 2.2 for programmers and non-programmers respectively. In addition, we asked programmers to rate their experience with the coding task. 40% rated it as difficult, 30% as neutral, and the rest as easy. Although all participants rated using WRAPI as easy, 80% of them indicated that it was difficult for them to find the required information in the API documentation. There is an education process for the users when it comes to dealing with APIs, and it depends a great deal on the quality of the API documentation [27].

Participant Feedback

Pros and cons of writing code to access APIs. We asked the programmer participants what they liked and did not like about writing code to access APIs. They mainly liked the flexibility and ability to integrate retrieved data into other applications and compute over it. Regarding dislikes, many participants indicated that dealing with pagination was a hassle, where one participant said "*I got stuck on the pagination and trying to figure out how to get 100 results*." Others stated that parsing the JSON response and retrieving specific fields was not easy, where one said "*It was not super easy to filter/debug the response object without constantly referring back to the documentation*." Finally, lack of familiarity with code made things difficult, with one person saying: "*Not knowing how to send the GET requests using Python*."

Pros and cons of using WRAPI. We asked all participants what they liked and did not like about WRAPI. One common answer for what they liked was accessing APIs without programming. One programmer said "*Much faster and simpler to use than simply coding to access the API*." A non-programmer said "*Everything about it is awesome! It's the type of tool needed to standardize API use across available open source datasets. I like the fact that once a user adds an API to scrAPIr, it's permanently added for other users to use*". Users also liked how the tool guided users through the integration process: "*ScrAPIr was able to cut down the number of steps significantly and is the first resource I used to guide my understanding of the entire process (other articles aren't so informative or easy to read)*". Finally, users enjoyed choosing response fields using the interactive pane and the automatic handling of the pagination. We noticed two main dislikes. First, the pagination step and how it gets mapped to SNAPI was confusing for some participants. Second, some programmers wanted a library using ScrAPIr as a proxy to access APIs and retrieve data directly in their code.

WRAPI versus writing code to access APIs. Programmers performed better using WRAPI than writing code for several reasons. First, they encountered challenges with code syntax: "*I could do everything with scrAPIr that I could do with code, but scrAPIr was faster and easier because I could avoid any issues related to syntax*". Programmers also took advantage of WRAPI's live feedback, automatic response, and pagination retrieval: "*it simplifies the trial/error through the simple validation process and allowed me to quickly search for fields without having to manually traverse the JSON outputs*". Finally, the guidance provided by WRAPI helped: "*Every API...uses different terms to describe endpoints, parameters, etc. ScrAPIr makes it easier to navigate through that maze by distilling required information into sections and providing common parameters as dropdowns*".

DISCUSSION

The long history of repurposing web data for mashups, analysis, and visualization shows that there is significant value in pulling data from websites. The spread of web APIs opens a powerful path to accessing this data by programming but this is effortful for programmers and impossible for non-programmers. Many users can easily *query* websites using the search forms those sites provide, but too many of those websites deliver those results only in a messy format that must be copied or scraped. Instead, we show that with Scrapir, just one enthusiastic user can do a limited amount of work to describe a large variety of APIs with HAAPI by reading the documentation. Once they have done so, anyone can easily use SNAPI to query and retrieve that API's data.

In a perfect world, the ScrAPIr ecosystem would not be necessary. If one of the API standardization efforts such as the Semantic Web or GraphQL were to succeed, then a tool like SNAPI could be created that would be able to query every web data source with no additional programming. But this kind of perfection requires cooperation from every website developer in the world. While many of these developers have demonstrated a willingness to provide their data through *some* idiosyncratic API, there is less sign of enthusiasm for following the constraints of an API standard. Thus, we instead aim to support other users who have the *incentive* to download the data.

LIMITATIONS AND FUTURE WORK

One limitation of ScrAPIr is that it only works on sites offering an API. As many sites do not, scraping will continue to be a necessary evil. However, the availability of APIs is increasing [32], and when they exist, ScrAPIr can gather higher quality data more easily than scraping [6]. ScrAPIr only offers basic filtering and sorting (select, project, and order by), without the richer "join" queries that databases or query languages like GraphQL can provide. But this parallels most websites that similarly offer limited query functionality. If users can extract data effectively, they can feed it into tools that provide richer query interfaces and visualizations.

Our study showed that Scrapir is easy to use to download data, but some programmers preferred to pull this data directly into their applications. Thus, we are currently building a library for ScrAPIr that lets users specify the API they want to access, filter conditions, number of results, fields from results, and the format they want results in (CSV or JSON). ScrAPIr will then handle the API request, parse the results, handle pagination, and return the results.

YouTube's API shows an important limitation of our approach. The API permits the use of special "operators", such as a minus-sign to indicate terms to exclude, within the textual query string. Our API description can only describe the query string as text; it has no way to describe the special semantics used to interpret that text. If instead the API offered separate "includeTerms" and "excludeTerms" parameters, our description could represent them meaningfully. Depending on the prevalence of this phenomenon, it might be worth expanding the description language to describe ways to assemble "structured" parameters from smaller pieces.

At present, ScrAPIr is not well designed for users who want different representations of the same API. If one user decides to add an API, they can select their desired parameters and response fields. But another user interested in querying that API may desire other parameters not included by the original user. Currently, we allow users to create multiple versions of the same API if they wish to add different parameters, but ideally this should be handled as one API with different versions. We will investigate how to maintain different customizations of the same API for multiple users.

CONCLUSION

This paper proposes the idea of designing an API description ontology that can be used to generate data query interfaces for general use. It presents ScrAPIr, a system that helps users access and query web APIs without programming, by providing a query interface over web APIs. Users can also publish these APIs for others to query them. Our user study showed that non-programmers can access APIs using ScrAPIr, while programmers can access APIs 3.8 times faster on average using ScrAPIr than writing code.

REFERENCES

[1] Apify. 2019. Apify Web Scraper. (2019). Retrieved November 17, 2019 from `https://apify.com/apify/web-scraper`

[2] Arvind Arasu and Hector Garcia-Molina. 2003. Extracting structured data from web pages. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 337–348.

[3] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *The semantic web*. Springer, 722–735.

[4] Tim Berners-Lee, James Hendler, Ora Lassila, and others. 2001. The semantic web. *Scientific american* 284, 5 (2001), 28–37.

[5] API Blueprint. 2019. (2019). Retrieved August 10, 2019 from `https://apiblueprint.org`

[6] Kerry Shih-Ping Chang and Brad A Myers. 2017. Gneiss: spreadsheet programming using structured web service data. *Journal of Visual Languages & Computing* 39 (2017), 41–50.

[7] Kerry Shih-Ping Chang, Brad A Myers, Gene M Cahill, Soumya Simanta, Edwin Morris, and Grace Lewis. 2013. A plug-in architecture for connecting to new data sources on mobile devices. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE, 51–58.

[8] Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *The 31st Annual ACM Symposium on User Interface Software and Technology*. ACM, 963–975.

[9] Roberto Chinnici, M Gudgin, JJ Moreau, and S Weerawarana. 2004. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3C Working Draft. *World Wide Web Consortium* (2004).

[10] Marco Cremaschi and Flavio De Paoli. 2017. Toward automatic semantic API descriptions to support services composition. In *European Conference on Service-Oriented and Cloud Computing*. Springer, 159–167.

[11] Jos de Jong. 2019. JsonEditor. (2019). Retrieved June 1, 2019 from `http://jsoneditoronline.org`

[12] Li Ding, Tim Finin, Anupam Joshi, Rong Pan, R Scott Cost, Yun Peng, Pavan Reddivari, Vishal Doshi, and Joel Sachs. 2004. Swoogle: a search and metadata engine for the semantic web. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*. ACM, 652–659.

[13] GraphQL. 2019. GraphQL. (2019). Retrieved August 1, 2019 from `https://graphql.org`

[14] Apis Guru. 2019a. OpenAPI Directory. (2019). Retrieved June 1 from `https://apis.guru/openapi-directory/`

[15] GraphQL APIs Guru. 2019b. GraphQL Voyager. (2019). Retrieved March 31, 2019 from `https://apis.guru/graphql-voyager`

[16] Marc Hadley. 2009. Web application description language. *World Wide Web Consortium Member Submission SUBM-wadl-20090831* (2009).

[17] Björn Hartmann, Leslie Wu, Kevin Collins, and Scott R Klemmer. 2007. Programming by a sample: rapidly creating web applications with d. mix. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM, 241–250.

[18] Patrick Hoefler, Michael Granitzer, Eduardo E Veas, and Christin Seifert. 2014. Linked Data Query Wizard: A Novel Interface for Accessing SPARQL Endpoints. In *LDOW*.

[19] Adrian Holovaty. 2005. ChicagoCrime. org. *Available at http* (2005).

[20] Mohammed Kayed and Chia-Hui Chang. 2010. FiVaTech: Page-level web data extraction from template pages. *IEEE transactions on knowledge and data engineering* 22, 2 (2010), 249–263.

[21] Nicholas Kushmerick. 2000. Wrapper induction: Efficiency and expressiveness. *Artificial intelligence* 118, 1-2 (2000), 15–68.

[22] Markus Lanthaler and Christian Gütl. 2012. On using JSON-LD to create evolvable RESTful services. In *Proceedings of the Third International Workshop on RESTful Design*. ACM, 25–32.

[23] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A Lau. 2009. End-user programming of mashups with vegemite. In *Proceedings of the 14th international conference on Intelligent user interfaces*. ACM, 97–106.

[24] Greg Little, Tessa A Lau, Allen Cypher, James Lin, Eben M Haber, and Eser Kandogan. 2007. Koala: capture, share, automate, personalize business processes on the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 943–946.

[25] Ben McIntyre. 2019. SlickGrid. (2019). Retrieved June 1, 2019 from `http://slickgrid.net`

[26] Lauren Murphy, Mary Beth Kery, Oluwatosin Alliyu, Andrew Macvean, and Brad A Myers. 2018. API Designers in the Field: Design Practices and Challenges for Creating Usable APIs. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 249–258.

[27] Brad A Myers and Jeffrey Stylos. 2016. Improving API usability. *Commun. ACM* 59, 6 (2016), 62–69.

[28] Bonnie A Nardi. 1993. *A small matter of programming: perspectives on end user computing*. MIT press.

[29] Node-RED. 2013. (2013). Retrieved August 15, 2019 from `https://nodered.org`

[30] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. 2013. An empirical study of API usability. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 5–14.

[31] Postman. 2019. Postman. (2019). Retrieved August 10, 2019 from `https://www.getpostman.com`

[32] ProgrammableWeb. 2005. ProgrammableWeb Search Category. (2005). Retrieved September 1, 2019 from `https://www.programmableweb.com`

[33] Mark Pruett. 2007. *Yahoo! pipes*. O'Reilly.

[34] Paul Rademacher. 2005. HousingMaps. *Available at http://www.housingmaps.com* (2005).

[35] RAML. 2019. RAML. (2019). Retrieved August 10, 2019 from `https://raml.org`

[36] RapidAPI. 2019. RapidAPI. (2019). Retrieved November 17, 2019 from `https://rapidapi.com`

[37] Leonard Richardson. 2007. Beautiful soup documentation. *April* (2007).

[38] Christopher Scaffidi. 2006. Why are APIs difficult to learn and use? *Crossroads* 12, 4 (2006), 4–4.

[39] A Scrapy. 2016. Fast and powerful scraping and web crawling framework. *Scrapy. org. Np* (2016).

[40] Swagger Specification. 2019. OpenAPI Specification. (2019). Retrieved August 15, 2019 from `https://swagger.io/specification`

[41] Swagger. 2019. Swagger. (2019). Retrieved August 10, 2019 from `https://swagger.io`

[42] TimeYourWeb. 2019. TimeYourWeb. (2019). Retrieved March 31, 2019 from `https://www.timeyourweb.com`

[43] Max Van Kleek, Daniel A Smith, Heather S Packer, Jim Skinner, and Nigel R Shadbolt. 2013. Carpé data: supporting serendipitous data integration in personal information management. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2339–2348.

[44] Jeffrey Wong and Jason I Hong. 2007. Making mashups with marmite: towards end-user programming for the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 1435–1444.

[45] Minhaz Zibran. 2008. What makes APIs difficult to use. *International Journal of Computer Science and Network Security* 8, 4 (2008), 255–261.