Systems to Democratize and Standardize Access to Web APIs

Tarfah Alrashed
MIT CSAIL
Cambridge, MA, USA
tarfah@mit.edu

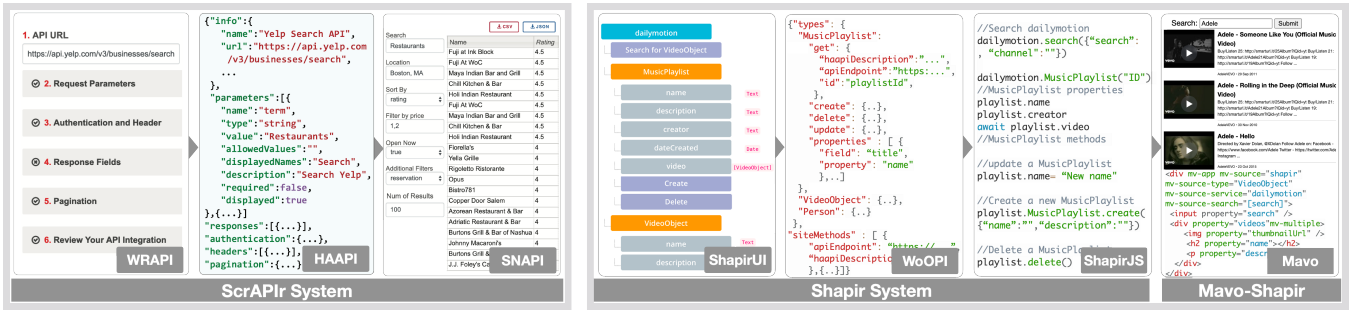


Figure 1: Systems to democratize and standardize access to web APIs. ScrAPIr enables users to query and retrieve data from web APIs without programming, and Shapir simplifies building interactive Web application that operates on web APIs.

ABSTRACT

Today, many web sites offer third-party access to their data through web APIs. However, manually encoding URLs with arbitrary endpoints, parameters, authentication handshakes, and pagination, among other things, makes API use challenging and laborious for programmers, and untenable for novices. In addition, each API offers its own idiosyncratic data model, properties, and methods that a new user must learn, even when the sites manage the same common types of information as many others. In my research, I show how working with web APIs can be dramatically simplified by describing the APIs using a standardized, machine-readable ontology, and building systems that democratize and standardize access to these APIs. Specifically, I focus on: 1) systems to enable users to query and retrieve data through APIs without programming and 2) systems that standardize access to APIs and simplify the work for users—even non-programmers—to create interactive web applications that operate on data accessible through arbitrary APIs.

CCS CONCEPTS

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

KEYWORDS

Web APIs, API Description Languages, Semantic Web, Schema.org, Data Standardization, Data Integration, Web Application

UIST '21 Adjunct, October 10–14, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8655-5/21/10.
https://doi.org/10.1145/3474349.3477587

ACM Reference Format:
Tarfah Alrashed. 2021. Systems to Democratize and Standardize Access to Web APIs. In *The Adjunct Publication of the 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21 Adjunct), October 10–14, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3474349.3477587

INTRODUCTION

Nowadays, substantial amounts of valuable data on many web sites can be accessed through web APIs (Application Programming Interfaces). Using these APIs, a programmer can create new applications that present and manipulate the data on those web sites in new ways. But using these APIs is a significant effort, even for skilled programmers. Data from the application must be marshalled and unmarshalled for delivery and proper API invocation URLs need to be generated. While skilled programmers may be familiar with this process in general, many newer and non-programmers will be baffled by the complexity of API usage [7]. In addition, each API is different, so even an author skilled in API usage must invest significant time reading documentation to learn any API they intend to use [4]. In our lab experiments, we found that the time required to learn and code to an API can be significantly larger than the rest of the time needed to create a simple application.

In my thesis, I introduce two sets of systems that simplify the work with web APIs by standardizing and democratizing access to these APIs.

- The first set of systems are part of the *ScrAPIr* ecosystem[1] [1], which enables end users to query and retrieve data from APIs without programming, by providing three connected components: 1) A standard ontology for describing an API in enough detail to automatically build a graphical interface (GUI) to query it, without per-API programming (HAAPI), 2) a tool that creates such a GUI from any HAAPI description. And 3) a form-based

[1] http://scrapir.org

tool that lets even non-programmers author HAAPI descriptions of web APIs.

- The second set of systems are part of the *Shapir* ecosystem[2] [2], which allows a user, using only GUIs and HTML authoring, to build a complete web application aggregating and interacting with data provided by multiple APIs. Shapir provides three related components: 1) A standardized, machine-readable API ontology (WoOPI), which can be used to provide a description that wraps the API with objects conforming to the *canonical type definitions* provided by Schema.org [5]. 2) A JavaScript library that uses a WoOPI description to present the API's data as typed objects in the local environment. And 3) a graphical tool that lets even non-programmers create the required WoOPI descriptions. We integrated Shapir with Mavo [9, 10], an interactive declarative HTML-based language, to empower a user to create applications interacting with APIs' data by writing only HTML, with no JavaScript programming required.

THESIS STATEMENT

We can design standardized, machine-readable API descriptions and build systems to enable users to access and manage web APIs' data, and the use of these systems will empower users, even non-programmers, to make use of web data in new and better ways.

SYSTEMS

Enable Data Query and Retrieval Through APIs

A common practice on the web is to extract and repurpose structured data from web sites. As far back as 2005, the HousingMaps and Chicagocrime *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 [3]. But, 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.

While programmers can and do use these non-programmer tactics, they also find ever more sites offering *web APIs*. 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 APIs also offer advanced searches and provide more extensive data than can be accessed by scraping webpages [3].

However, a survey we conducted found three significant obstacles to using APIs: 1) People need to write code to access APIs, making them inaccessible to non-programmers, 2) Even for programmers, APIs can be hard to learn and use[7], and 3) Most modern APIs 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.

[2]http://shapir.org

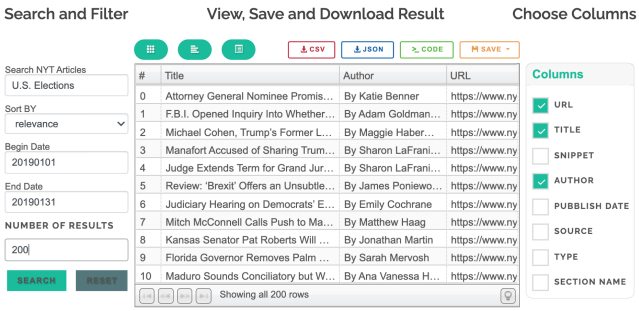


Figure 2: An illustration of SNAPI for the NY Times API

To simplify API programming, efforts have been made to design *API description languages* [1] that provide structured descriptions of web APIs. The machine-readable descriptions can be used to generate documentation for programmers and code stubs for connecting to the APIs through various languages. The most popular at present is the OpenAPI specification [8]. 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.

To make these web APIs accessible to everyone, a query interface that allows users to query and download APIs' data, and hides the complexity of these APIs is necessary. Building a query interface for a couple of APIs can be manageable, only by programmers of course. But there are tens of thousands of web APIs and these APIs are different [6]. In addition, both programmers and non-programmers need these APIs' data. Creating a query interface on top of every API is just infeasible.

A solution to this problem is to automate the process of creating a query interface for every web API. Although these APIs are different, many of them are assembled from a small palette of possible design choices. If we can enable users, programmers and non-programmers, to describe those design choices in a machine-readable fashion, then it would be possible to automatically generate a query interface on top of these APIs and allow anyone to query them.

To make this possible, we built *ScrAPIr*[1], an ecosystem that empowers end users to query and retrieve data from APIs without programming. ScrAPIr consists of three related components: HAAPI, WRAPI, and SNAPI. HAAPI is a standard ontology for describing an API in enough detail to automatically build a graphical interface (GUI) to query it, without per-API programming. HAAPI extends the OpenAPI specification and describes the API's query parameters (used to filter objects), their types, the authentication and pagination methods (if any), and the structure of the returned data. SNAPI is a search GUI 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 query0, invokes the relevant API, and shows results in a tabular (spreadsheet) or other standard data format (Figure 2). And WRAPI is a tool 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. These three components are connected, as shown in Figure 1 (left).

Any user can first use WRAPI to author a HAAPI description of any web 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 conducted user studies providing evidence that it is easy for users to use the (quite typical) SNAPI search form to retrieve data from HAAPI descriptions. More significantly, we also find that novices 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.

In addition, we evaluated how well the HAAPI ontology is able to describe query-based APIs. We sampled the ProgrammableWeb [6] list of available web APIs and found that 97% (with a 95% confidence interval of 0.93 ± 0.062) of search APIs can be described by HAAPI such that SNAPI can query them.

We extended ScrAPIr to support describing APIs that modify data, empowering end users to edit web data through APIs without programming. In addition, we built a library for ScrAPIr that lets programmers access APIs, described using HAAPI, and retrieve/-manage these APIs' data directly in their code.

Standardize and Democratize Access to Web APIs

While ScrAPIr provides tools to enable users to access and interact with web APIs, it only allows users to describe individual API endpoints, ScrAPIr thus does not support data integration within the same or different APIs. In addition, ScrAPIr does not standardize the data models of web APIs, building an application using one API will not work with another API offering the same types of data.

Consider a user who wants to create a small application for collecting videos they find on the web, organizing them into playlists, and playing them. They likely start with a mental model of their data: videos with titles, creators, creation dates, and video data links; and playlists, each with a title, a creation time, and a collection of videos in the playlist. It would be relatively straightforward to create a basic web application for managing this data model, presenting forms that allow the author to view and edit the information about each video and move videos among playlists. Indeed, with an HTML templating tool like Mavo [10] the author would not even really need to write any JavaScript code: they would create an HTML document that looks like the desired application, then add a small amount of Mavo markup which indicates which elements of the HTML are editable data, and the Mavo library would read those annotations and provide the relevant data editing, presentation, and storage capabilities.

But suppose the user wanted to enrich their application by enabling it to search for videos and playlists on Dailymotion, and pull the resulting information into their application to manage it? Now the task becomes significantly harder. ScrAPIr can help the user to search Dailymotion for videos or playlist, but the user will have to write JavaScript code *connecting* the playlists with their videos. Then they would need to write more code translate the data coming back from Dailymotion to match the schema they have chosen for their application (and to translate back if they are sending updates in the opposite direction), as it is unlikely that Dailymotion has selected the same property names and values as they did.

If the user then decided to incorporate Vimeo videos into their application as well, they would have to repeat the entire process: learn an entirely different API and translate those results (using a different dictionary) into their own preferred schema.

This task demands a significant amount of labor with little creative or intellectual content. The user knows from the beginning that these sites have videos and playlists, but must labor to learn about and translate between multiple inconsistent website API syntaxes and data models. It takes the user far away from their initial simple model of video and playlist objects with readable and writeable properties, and the simple application they build with elementary programming (or, if using Mavo, writing nothing but HTML).

To eliminate much of this labor overhead, we designed a new API description language, the *Web of Objects Programming Interface* (WoOPI), a simple schema for (i) modeling an API's data as a collection of typed objects with read/write properties and methods, and (ii) describing how to implement that model via appropriate calls to the site's API. We designed WoOPI by analyzing a random sample of web APIs from the ProgrammableWeb to understand what was common among them. We then built *Shapir*[2], an ecosystem that significantly simplifies the work for users—even non-programmers—to create interactive web applications that operate on standardized data accessible through arbitrary web APIs.

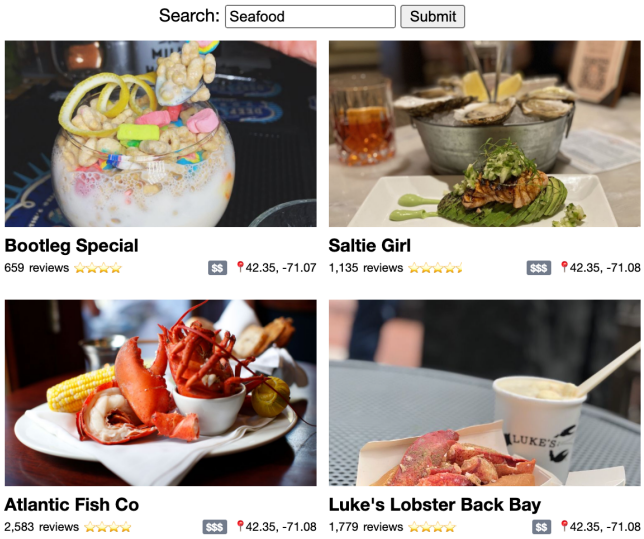
Shapir (Figure 1 (Right)) provides a graphical interface (ShapirUI) that permits any user (even a non-programmer) to describe any web API using a WoOPI ontology that maps the website objects to *standard data types and methods* found on Schema.org [5], which offers a common set of schemas for describing objects on the web and is supported by major search engines. Given such a WoOPI description, Shapir automatically generates a JavaScript library (ShapirJS) that presents all the API objects as objects in the application's local

```
// Get the playlist
let playlist = await vimeo.MusicPlaylist("8274189");
// Read playlist information
console.log(playlist.name, playlist.description);
// Get the playlist videos
let videos = await playlist.video;
// Get the comments of the 8th video
let videoComments = await videos[7].comment;
// Search Vimeo
videos = await vimeo.search("Adele",{sort:"relevant",
     filter:"trending", numberOfItems:100});
// Create a new playlist
playlist = await vimeo.MusicPlaylist.create({
     name:"New", description:"New", layout:"player"});
// Update the playlist's description
playlist.description = "Still New";
// Delete the playlist
playlist.delete();
```

Figure 3: An example of the Vimeo API with ShapirJS

environment, which can be manipulated by getting and setting object properties or invoking apparently-local methods. Listing 3 shows a ShapirJS code snippet for accessing the Vimeo API via a WoOPI that maps to Schema.org `MusicPlaylist` and `VideoObject` types. `MusicPlaylist` is a Schema.org type that returns an object of type MusicPlaylist which includes `name`, `description` and `video` properties. The `video` property of `MusicPlaylist` is a collection of `VideoObject` objects. Every `VideoObject` has a `comment` property that returns an array of `Comment` objects. Users can search Vimeo using the `vimeo.search()` method. All objects returned are "live", meaning developers can create, delete, and update `MusicPlaylist` objects by manipulating the array returned or its contents.

With ShapirJS, a programmer unfamiliar with APIs can author their applications as if the data they are manipulating is already in their hands. Because the provided data types fit the Schema.org standard, an application written over one API will work, unchanged, for any other API providing semantically-equivalent data.




```
<div mv-app mv-source="shapir" mv-source-search="[search]"
  mv-source-service="yelp, foursquare">
    <input property="search" />
    <div property="businesses" mv-multiple>
        <img property="image" />
        <h2 property="name"></h2>
        <span property="reviewCount"></span> reviews
        <span property="aggregateRating"></span>
        <span property="priceRange"></span>
        <span property="latitude"></span>,
        <span property="longitude"></span>
    </div>
</div>
```

Figure 4: A Mavo application that integrates and displays standardized data from Yelp and Foursquare APIs

We integrated our JavaScript library with Mavo making it possible for users to create standalone web applications that manipulate data over (multiple) web APIs without writing a single line of JavaScript. Listing 4 shows a Mavo application to search both Yelp and Foursquare and list their restaurants.

We evaluate the success of these components through a series of user studies: one in which users use ShapirUI to create WoOPI descriptions, another in which users program simple web applications in JavaScript over the WoOPI-generated library, and a third in which users write HTML to create Mavo applications that interact with the websites' data using the WoOPI description. From our user studies, we found that programmers are able to create simple data management applications that require multiple HTTP requests 5.6 times faster on average using the Shapir generic library than using the popular Swagger API integration library[3]. Using our Mavo integration, non-programmers were able to build functioning data management applications that access multiple web APIs in just 4 minutes. In addition, we evaluated WoOPI against a random sample of 60 APIs, and we found that 90% (with a 95% confidence interval of 0.90 ± 0.076) of these APIs can be fully described by WoOPI such that a ShapirJS library can be generated to query them and manipulate their objects.

CONCLUSION

My dissertation presents systems and ontologies for supporting and simplifying the access and use of web APIs. Specifically, my work supports: *Data query and retrieval*—any user can query and download APIs' data without programming; *Reusability*—a user can build one application for a Schema.org type that would work with any API exposing that type; and *Data integration*—the user can pull semantically-equivalent data from multiple APIs in one application. And *learnability*, no need to learn additional APIs. Through this research, I hope to empower more people to make use of the valuable data provided by web APIs.

ACKNOWLEDGMENTS

I am grateful to my advisor David Karger for his continual guidance and feedback. This work was supported in part by KACST and MIT.

REFERENCES

[1] 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.
[2] Tarfah Alrashed, Lea Verou, and David R Karger. 2021. Shapir: Standardizing and Democratizing Access to Web APIs. Conditionally Accepted at UIST 2021: ACM Conference on User Interface Systems and Technology Symposium.
[3] 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.
[4] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. 2014. How do API documentation and static typing affect API usability?. In *Proceedings of the 36th International Conference on Software Engineering*. 632–642.
[5] Ramanathan V Guha, Dan Brickley, and Steve Macbeth. 2016. Schema. org: evolution of structured data on the web. *Commun. ACM* 59, 2 (2016), 44–51.
[6] ProgrammableWeb. 2005. ProgrammableWeb Search Category. Retrieved September 1, 2019 from https://www.programmableweb.com
[7] Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009), 27–34.
[8] Swagger Specification. 2019. OpenAPI Specification. Retrieved August 15, 2019 from https://swagger.io/specification
[9] Lea Verou, Tarfah Alrashed, and David Karger. 2018. Extending a Reactive Expression Language with Data Update Actions for End-User Application Authoring. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 379–387.
[10] Lea Verou, Amy X Zhang, and David R Karger. 2016. Mavo: creating interactive data-driven web applications by authoring HTML. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. ACM, 483–496.

[3] https://github.com/swagger-api/swagger-js