Shapir: Standardizing and Democratizing Access to Web APIs

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

Lea Verou
MIT CSAIL
Cambridge, USA
leaverou@mit.edu

David R. Karger
MIT CSAIL
Cambridge, USA
karger@mit.edu

ABSTRACT

Today, many web sites offer third-party access to their data through web APIs. But 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 site 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 this work, we show how working with web APIs can be dramatically simplified by describing the APIs using a standardized, machine-readable ontology. By surveying a statistical sample of web APIs, we develop a simple ontology that can effectively describe the core functionality of nearly all of them. We then present Shapir, a system that includes a graphical, form-based authoring tool for the API description, from which Shapir can automatically generate a standardized JavaScript library for accessing data on the web site as objects with readable and writeable properties. This enables programmers to access data without learning the details of each API, and indeed allows them to use the same unchanged code for multiple web sites. We then integrate Shapir with Mavo, an HTML language extension for describing web applications declaratively, to also empower plain-HTML authors to access these APIs. In our lab evaluation, we found that programmers are able to accomplish program data management tasks that require multiple API requests 5.6 times faster on average using the Shapir generic library than using the popular Swagger API integration library. Using our Mavo-Shapir integration, even non-programmers were able to build functioning data management applications that access multiple web APIs in just 4 minutes.

CCS CONCEPTS

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

KEYWORDS

Web APIs, API Description Languages, Schema.org, Semantic Web

ACM Reference Format:
Tarfah Alrashed, Lea Verou, and David R. Karger. 2021. Shapir: Standardizing and Democratizing Access to Web APIs. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21), October 10–14, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/3472749.3474822

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

ACM ISBN 978-1-4503-8635-7/21/10.
https://doi.org/10.1145/3472749.3474822

1 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 web API usage [36]. In addition, each API is different, so even a programmer skilled in API usage must invest significant time reading documentation to learn any API they intend to use [12]. As we show experimentally below, 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 this work we describe Shapir[1], an ecosystem that significantly simplifies the creation of interactive web applications that operate on data accessible through arbitrary web APIs. Shapir provides a graphical interface that permits any user (even a non-programmer) to describe any web API using a *Web of Objects Programming Interface* (WoOPI) ontology that maps the web site objects to *standard data types and methods* found on Schema.org [17], a site that 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 can automatically generate a client library that presents the web site data as objects in the application's local environment, which can be manipulated by getting and setting object properties or invoking apparently-local methods. Using this library, 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 website will work, unchanged, for any other website providing semantically-equivalent data. Shapir also integrates with Mavo [43], a library that empowers non-programmers to create interactive web applications simply by authoring HTML. Combined, Shapir and Mavo make it possible to create standalone web applications that manipulate data, even over (multiple) web APIs simultaneously, without writing a single line of JavaScript.

The Shapir ecosystem is based on a simple standard description language, which makes it modular and distributed. WoOPI API descriptions can be published anywhere, and the small ShapirJS library that leverages those descriptions can run anywhere. Our approach involves no bottleneck platform or server, leaving people

[1] https://shapir.org

free to use it anywhere, and to modify, remix, or replace individual components as they see fit.

1.1 Motivation

Consider someone 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 Mavo [43] the author would not even really need to program: 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. 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. To begin, they would need to learn the whole concept of web APIs—the idea that you can generate specially formatted HTTP requests that fetch data from or modify data on websites. They would then need to study the Dailymotion API documentation and determine how to write the appropriate HTTP requests to fetch that data, and the JavaScript necessary to unpack what is returned. 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, generate appropriate HTTP requests to it, unpack the returned data, and translate those results (using a different dictionary) into their own preferred schema.

This task demands a significant amount of tedious labor. The user knows from the beginning that these sites have videos and playlists, but must work 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).

1.2 Our Approach

In this work, we propose an approach to eliminating much of the mental and manual labor overhead of working with web APIs. We propose a new API description language, the Web of Objects Programming Interface (WoOPI), that can be used to describe most existing website APIs. We then provide a JavaScript library (ShapirJS) that uses the WoOPI description to provide proxy objects in the local programming environment for each object available through the website API. A user can read and write properties of those objects, and invoke methods on them, as if they were local, and the library takes care of making the necessary API calls (as described in WoOPI) to provide or modify the relevant data.

Equally important, WoOPI can wrap the API with objects conforming to *canonical type definitions* provided by Schema.org. For example, both Dailymotion and Vimeo videos in the example above can be wrapped in objects conforming to the `Video` type from Schema.org, with canonical properties such as `creator`, `dateCreated`, and `name`. Thus, the same code that the user writes to incorporate videos from Dailymotion into their application will work *unchanged* to incorporate videos from Vimeo, or from any other website with a suitable WoOPI description.

In addition, we have incorporated the WoOPI interpreter into the Mavo framework. A user can thus construct their video management application entirely by authoring HTML, and then direct this application (still just HTML) to retrieve and incorporate video information from both Vimeo and Dailymotion.

This scenario relies on the existence of WoOPI descriptions for specific websites. Our final contribution is ShapirUI, a graphical, form-based authoring tool for WoOPI descriptions. ShapirUI steers users through a process of *describing* the API and its alignment with standard Schema.org types and methods, all without writing any code.

In summary, in order to simplify interaction with data on web sites, we offer the following contributions:

(1) WoOPI, a simple schema for (i) modeling a website's data as a collection of typed objects with read/write properties and methods using (ii) *canonical data types* from the Schema.org standard, and (iii) describing how to implement that model via appropriate calls to the site's API.
(2) Evaluation of a random sample of web APIs, showing that WoOPI suffices for describing roughly 90% of those APIs.
(3) ShapirJS, a JavaScript library that uses a WoOPI description to present the website's data as typed objects in the local environment.
(4) ShapirUI, a graphical tool that lets even non-programmers create the required WoOPI descriptions.
(5) Integration with Mavo, which allows a user to create applications interacting with web-site data by writing only HTML, with no JavaScript programming required.

Taken together, these components (Figure 1) empower a user, using only GUIs and HTML authoring, to build a complete web application aggregating and interacting with data provided by multiple web APIs.

We evaluate the effectiveness of this approach through a series of user studies: one in which users use ShapirUI to create WoOPI descriptions, another in which programmers create simple web applications in JavaScript using the WoOPI-driven ShapirJS library, and a third in which users (including non-programmer HTML authors) write HTML to create Mavo applications that interact with the websites' data using the WoOPI description.

Languages and Components before Platforms. Shapir is a collection of small interoperable ontologies and systems, rather than a monolithic platform. This decomposition offers meaningful

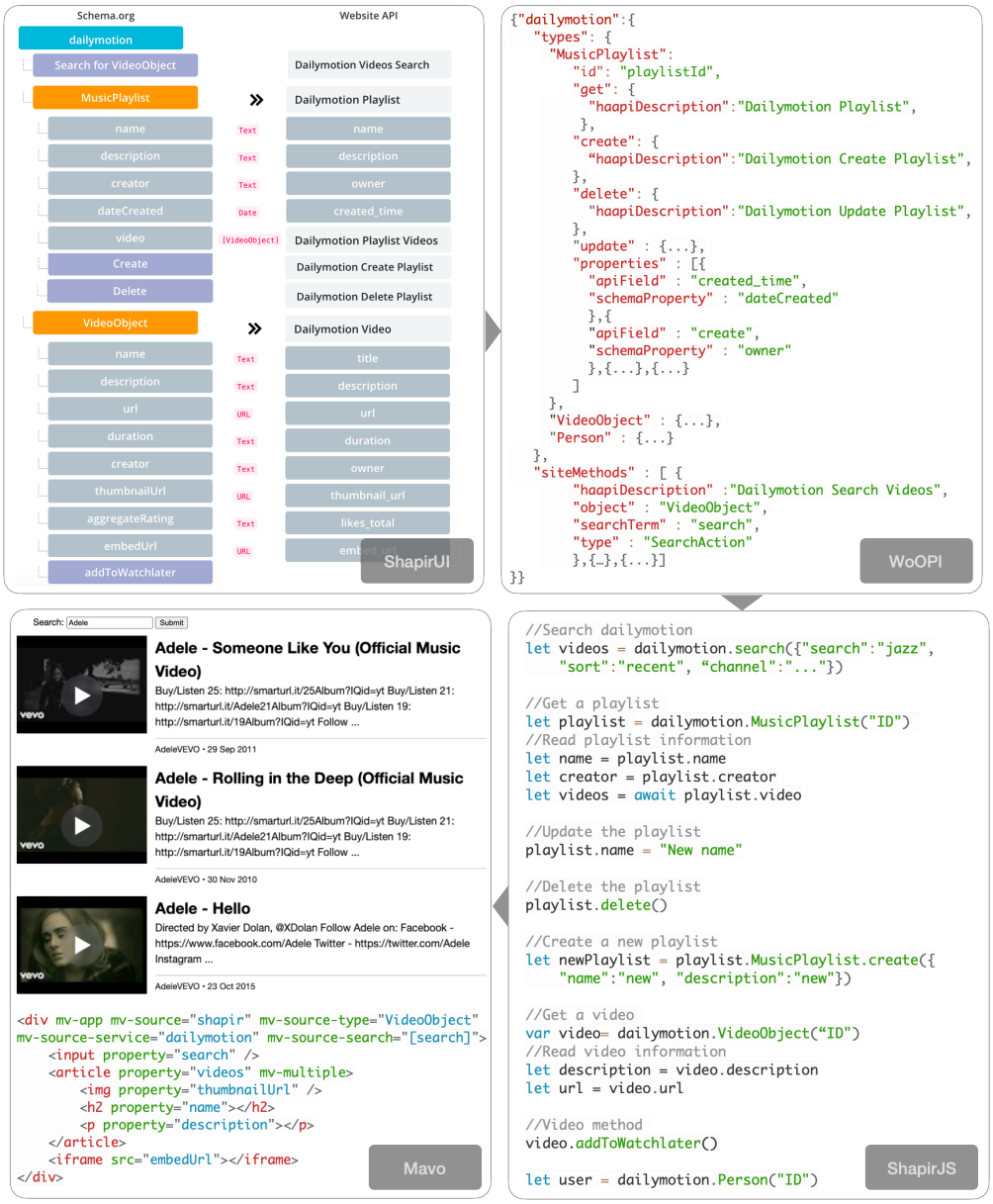


Figure 1: An illustration of the Shapir system, consisting of the components ShapirUI, WoOPI, ShapirJS and its integration with Mavo, being used to create a small video playlist app.

modularity benefits. Our pipeline from web APIs to Mavo applications demonstrates one powerful combination of these components, but there are many others. The standard-typed objects provided by ShapirJS using WoOPI can be useful in a variety of programming tasks, without any reference to Mavo. WoOPI can also clearly be used to generate class definitions for other programming languages, or be interpreted by other web frameworks such as React or Vue.js to automatically generate components for interacting with data behind wrapped APIs. WoOPI could also be used to declare object mappings for APIs described by other ontologies, such as WSDL. Other, better tools for generating WoOPI descriptions could coexist comfortably with ShapirUI. The different layers of our decomposition offer useful functionalities on their own but have small surfaces that make it easy to replace them with better alternatives in the future.

Equally important, our work is achieved through *declarative languages* that can be standardized and used by a variety of competing but interoperating tools. Such an outcome would offer far more diversity and freedom to innovate than the walled-garden API integration platforms currently being developed (see Section 2.5).

Our proposal, that a small declarative schema suffices for API integration and interoperability, is another part of our contribution.

2 RELATED WORK

The plethora of inconsistent web APIs imposes a substantial burden on anyone hoping to interact with the data they offer. Several distinct approaches have been explored to reduce this burden. Most ambitious is the idea, typified in the Semantic Web and Open Graph Protocol [20], of replacing all web APIs with a *single, standardized protocol* for accessing data on any website. A less top-down approach is to *provide API description languages* that can be interpreted by clients to connect to the described APIs. More recently, we've begun to see the emergence of *API integration platforms* that proxy access to numerous APIs through a single meta-API.

Once data has been accessed, users need tools to interact with it, which has motivated a long line of work on *Mashups* that combine data from multiple (inconsistent) websites, and visual tools for creating applications to present and manipulate the data.

2.1 One Standard API

Having to learn and program to a new API for every website is challenging, 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 [3] proposed RDF, a single graph structured data model, and SPARQL, a standard query language that would work on any data repository. But the Semantic Web has faced adoption challenges [21]. Even more recently, GraphQL [16] has been put forward; it is too early to know how widely it will be adopted.

An alternative design is the Open Graph Protocol [20], Facebook's version of the Semantic Web and Linked Data, which standardizes the use of metadata within a web page to represent the content of a page. But outside of Facebook, the Open Graph Protocol also has not been widely adopted [14]. Open Graph is not the only web page semantic markup in use. Schema.org [17] is another one that offers a richer vocabulary than Open Graph to describe the content of web pages. Schema.org is mainly used by Search Engines to improve the user experience on their search results. We chose Schema.org for describing web APIs data because both search engines and open-source tools have used it successfully to build an open ecosystem for various types of content [5].

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.

2.2 API Description Languages

In the absence of a single standard API, many efforts have been made 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) [18], both using XML. More recently, the OpenAPI specification language, API Blueprint [4], RESTful API Modeling Language (RAML) [34], and Human accessible API (HAAPI) [1] have emerged.

Like WoOPI, these API descriptions describe technical aspects of APIs (input and output parameters, data types, etc). They have been used primarily in two ways: first, to automatically generate API connector code, and second, to automatically generate documentation describing the API to developers.

However, unlike WoOPI, these prior schemas describe APIs *as callable endpoints* but do not describe the data model exposed through those endpoints (objects, types, the IDs that connect these objects, how to read and write objects' properties, etc). WoOPI describes APIs at a higher level of abstraction which enables the automated generation of a typed local data model using ShapirJS. We show how this richer description improves both programmatic and declarative data access and significantly decreases the effort end users must invest to write applications that access APIs in standard ways.

Like ShapirUI, Swagger [38], Postman [30], RapidAPI [35], and HAAPI offer graphical, web-form-based editors that enable developers to describe their APIs. Because our underlying WoOPI ontology is richer in some dimensions than those others, we had to solve new problems in order to create a usable editor.

2.3 Mashup Tools

Kicked off by ChicagoCrime.org [22], a long line of research has explored ways to let end users create alternative visualizations and applications over data accessed through website APIs. The earliest mashups were hand-crafted by programmers. A later generation of tools, such as Marmite [44] and Carpé Data [41], helped users author the applications, but assumed the existence of pre-programmed web API connectors to access specific data sources, or for programmers to create new ones. D.mix [19] 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."

Gneiss [7] gives users a spreadsheet-like interface to build application query over any publicly accessible data APIs. But it expects users to do so by manually entering 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, or authentication. Mavo [43] is a library that lets a user build an interactive application over any JSON data source, simply by authoring and annotating an HTML document (we will use Mavo as part of our overall system). But like the other tools above, Mavo posits that programmers will provide the connectors to those JSON data sources.

Easing the burden further, Spinel [8] does help end-users connect data APIs to mobile applications without programming. Spinel has a form-based query-builder for describing an API. However, the resulting description is used to feed data to existing mobile applications, and Spinel's meta-schema for describing APIs is missing even more key elements than OpenAPI (e.g. authentication).

These prior tools focus more on building applications and mashups over easily accessible data. WoOPI descriptions would provide a way for all these past tools to read descriptions of new websites' API and integrate them into their workflow (this again

shows the benefit of designing a standard description language that can be used anywhere, rather than a platform). WoOPI's main focus is to make data easily accessible. In addition, Shapir allows relatively inexperienced programmers to describe and connect any web APIs using ShapirUI, without writing code or assembling any query strings. With Mavo, Shapir also allows non-programmers to create HTML applications to query and access these APIs.

All these prior tools focus on querying and retrieving API data; none of them support editing data through web APIs. Shapir supports read *and* write; it allows users to access and manipulate data through web APIs.

Data flow language tools such as Yahoo Pipes [32] and Node-RED [28] allow users to edit data through APIs. Node-RED allows users to configure individual HTTP requests, using a web form, and connect them together. Node-RED does not offer an easy way for end users to connect data from Node-RED to their applications. Users need to make changes to their application and write code to make this connection. In addition, studies have found that the data flow representation is often difficult for end-users to understand [6].

2.4 Data Integration

Several efforts have been made to help users integrate data from multiple sources. Research on mashup tools has provided ways to let users extract and integrate web data from different websites without having to write conventional code [10]. Mashup tools like MashMaker [13] and Vegemite [27] allow users to scrape web data directly from web pages, and allow users to create a mashup by browsing and combining different web pages. Potluck [23] is another data integration tool that allows users to combine, clean and merge data coming from different sources. However, the majority of these mashup tools do not support the reuse and integration of the created Mashups. In addition to mashup tools, the Semantic Web vision includes representations like RDFS and OWL that can be used to drive inference engines able to transform data between multiple schemas. RDFS aims to support transformation between any two schemas at any time. We choose a less ambitious but more practical approach of describing and executing a translation from specific website's API to the (single) Schema.org standard as we fetch the data, so that no transformations need to be applied during computation.

2.5 API Integration Platforms

Several platforms have been built to provide access to large numbers of web APIs. RapidAPI [35] and Prompt API [2], like Shapir, each offer a repository of APIs that programmers can access and integrate into their applications. They both provide users with a library and code snippets that allow users to access these APIs only through their servers and charge users to access their APIs or request more than 10 requests/day. Thus, these tools can become bottleneck: their servers control and must work to provide API access to their users. In addition, RapidAPI and Prompt API do not provide a way for novices and non-programmers to access their APIs. Shapir, on the other hand, offers a repository of web APIs "descriptions" and simplifies users' access to these APIs in a decentralized way. Shapir provides a local programming environment for users to interact directly with APIs through their WoOPI descriptions. Shapir also empowers novices and non-programmers to author WoOPI descriptions and access arbitrary web APIs.

Zapier [45] and IFTTT [24] also provide a repository of web APIs but unlike RapidAPI and Prompt API, they allow users to connect different APIs, and without coding. The goal of both IFTTT and Zapier frameworks is to provide an easy way for non-programmers to automate activities across multiple services by integrating their functionalities [33]. However, IFTTT and Zapier use very constrained workflows and control how users use their APIs. And although both are no-code tools, they again rely on pre-existing API connectors—they do not offer a way for users, even developers, to add new APIs. In contrast, anyone can use ShapirUI to describe any new API and use Mavo to build an application that implements IFTTT-like functionalities without its constraints.

3 BACKGROUND

Our system builds heavily on three pieces of prior work that we describe here: ScrAPIr, Mavo, and Schema.org.

3.1 ScrAPIr

ScrAPIr [1] is a system that empowers end-users to graphically query and retrieve data from web APIs. ScrAPIr presents a *Human Accessible API (HAAPI)* schema that describes web APIs in terms of the available API endpoints (functions) and the arguments to them, as well as the format of results. ScrAPIr includes a GUI for authoring HAAPI descriptions for a web API. Given such a HAAPI description, ScrAPIr can automatically generate an end-user interface that enables any user to query the described API and view and filter the returned results in a spreadsheet-like interface. HAAPI was originally focused on search APIs, but for this work we extend the HAAPI schema to support arbitrary API endpoints that retrieve or mutate data.

In this work we place a richer abstraction layer over HAAPI via a *Web of Objects Programming Interface (WoOPI)*. While HAAPI describes web sites in terms of opaque methods that accept and return primitive string or number values—which is sufficient to support a search interface—WoOPI describes the API in terms of *types* of objects managed, and type-specific methods that *create, read, update, and delete* those objects (and collections of them). This richer representation enables us to automate more and richer interactions with the API than was possible with HAAPI. An example of WoOPI and HAAPI markup is discussed in detail below in Figure 2.

WoOPI could be built on top of any low level API description (HAAPI, OpenAPI, RAML, WSDL, etc). But we chose HAAPI because it offers additional advantages over the other API description schemas. HAAPI is built on top of a lower level OpenAPI description [37] that describes endpoints, parameters, and response structure. HAAPI augments the OpenAPI description as follows: (1) HAAPI specifies information about how to authenticate to the API or paginate to gather large numbers of results. Authentication and pagination to be some of the main challenges programmers face in dealing with APIs [1]; Building WoOPI on top of HAAPI allowed us to automatically generate a standard API that takes care of the pagination and authentication for users. (2) HAAPI provides default values for all required parameters and some optional ones. This

helps users start querying the API quickly and iterate by adding more parameters, taking advantage of the feedback that the intermediate responses provide. If we used another description schema, we would need to augment it to handle these issues as well.

Our previous work [1] demonstrated that end users without programming experience could author HAAPI descriptions of API endpoints. Thus, in our current work (which also targets non-programmers) we posit the existence of a previously-created HAAPI description and demonstrate a GUI that can be used to augment that description with WoOPI information. ScrAPIr already offers a repository of HAAPI descriptions for more than 170 API endpoints, that can be easily accessed by Shapir.

3.2 Mavo

Mavo [42, 43] is a "bidirectional" HTML templating language that extends the declarative syntax of HTML to describe Web applications that manage, store and transform data. Mavo aligns hierarchically structured data to a hierarchically structured web page. Authors link a page to to a data source, then add a few attributes and expressions to their HTML elements to transform a static web page into a persistent, data-driven web application.

Mavo includes support for common storage APIs, such as Dropbox or GitHub, which it uses to read and store JSON files with the data for each Mavo app. Authors specify which of these predefined storage APIs they intend to use by using `mv-storage` (read-write) or `mv-source` (read-only) HTML attributes. Support for new APIs can be added by authoring JavaScript plugins.

For example, the HTML below defines a complete to-do list application whose data is stored on GitHub. Based on the markup, Mavo supports adding, deleting, and editing todo items in the list.

Loading data from arbitrary APIs in Mavo applications is technically already *possible*. Any URL that returns JSON is a valid `mv-source` for Mavo. However, this requires authors to manually assemble API-invocation URLs, and does not provide help with authentication or pagination, so it is of limited utility.

3.3 Schema.org

Schema.org [17] is an initiative by the world's biggest search engines (Google, Bing, Yahoo and Yandex), that provides a collection of schemas that webmasters can use to describe or mark up their web pages in ways recognized by these major search engines. Search Engines use these descriptions to enrich the user experience on their search results and to generate rich snippets. Schema.org is widely used all over the web [29] and made up of more than a 1000 attributes organized into a few primary types: Thing, Action, Creative Work, Event, Medical Entity, Organization, Person, Place, and Product. Subtypes include Article, Video, Image, Book, Movie, Restaurant, and Recipe. At present, Schema.org is being used primarily to mark up web pages with information helpful to search engines.

Here, we propose an additional application of schema.org, to describe data provided by web APIs and standardize access to these APIs for uses other than search. We choose Schema.org because it is a rich vocabulary that it is widely being used to describe content on web pages [17], which is similar to the content of these website's APIs.

Our approach could also enhance search. Current vertical search engines that rely on the presence of structured metadata (e.g., jobs, datasets, etc) on Web pages are not able to discover and index structured content that can only be accessed through web APIs. The availability and use of web APIs have increased in the last decade [11], and a number of these APIs provide access to valuable data that is not necessarily presented on web pages. Describing these APIs in using standard data types might allow search engines to discover and index these sites' data, making it more findable. The data behind APIs is also often cleaner and more complete than what can be extracted from web pages.

3.4 Unification

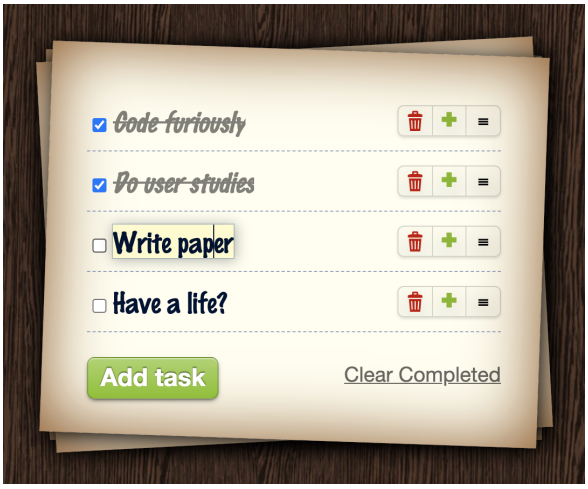
Shapir bridges the significant gap between two of the background systems: ScrAPIr and Mavo. Non-programmers are not equipped to pass parameters through chains of function calls, even if those function calls are standardized through HAAPI. But they can certainly understand the concept of objects with readable and writeable properties—the conceptual model underlying the Mavo HTML templating language. To bridge the gap, Shapir provides another graphical authoring tool (ShapirUI) for another ontology, WoOPI, which describes how to take an API of arbitrary methods and parameters (as described by HAAPI) and wrap it in a "normalized" API consisting of a collection of typed objects with readable and writeable properties. Our ShapirJS library reads an arbitrary WoOPI description and provides the declared objects within the browser's JavaScript environment where they become available for manipulation by Mavo.

In addition, incorporating Schema.org as a way to steer WoOPI descriptions toward common data types means that Shapir applications can be used unchanged over any of the web APIs that have been wrapped by those common data types, increasing portability and reuse.

4 THE SHAPIR ECOSYSTEM

Shapir is 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. It consists of three related components: WoOPI, ShapirJS, and ShapirUI. WoOPI is a standardized, machine-readable API ontology that can describe an API in terms of objects conforming to the *canonical type definitions* provided by Schema.org. ShapirJS is a JavaScript library that uses a WoOPI description to present the API's data as typed objects in the local environment. And ShapirUI is a graphical tool that lets even non-programmers create the required WoOPI descriptions, using standard data types. These three components are connected. A person uses ShapirUI to describe an API, and ShapirUI generates a corresponding WoOPI description of it. The ShapirJS JavaScript client library can read that WoOPI description to provide simple, local-environment access to the data behind the API. The WoOPI description only needs to be authored once; then anyone can use it. We also integrated ShapirJS with Mavo, 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.

```
<!-- Mavo App -->
<ul mv-app mv-storage="https://github.com/janedoe/todos">
    <li property="task" mv-multiple>
        <label>
            <input property="done" type="checkbox" />
            <span property="taskTitle">Do stuff</span>
        </label>
    </li>
    <button mv-action="delete(task where done)">
        Clear Completed
    </button>
</ul>
<!-- JSON data from GitHub -->
{"task":[
    {"taskTitle":"Code furiously", "done":true},
    {"taskTitle":"Do user studies", "done":true},
    {"taskTitle":"Write paper", "done":false},
    {"taskTitle":"Have a life?", "done":false}
]}
```



5 WOOPI: A WEB OF OBJECTS PROGRAMMING INTERFACE

As discussed in the introduction, our goal is to expose website data to users as no different from the data a user would work with in their own local environment. In particular, we wish to present the data as

- *objects* of different *types*,
- each with specific *properties* that a user can *get* and *set*,
- which can be *added to* or *removed from* various *collections*,
- and possibly with *methods* that can be invoked on objects of a given type.

In contrast, web APIs are generally presented via *endpoints* that are invoked over HTTP; each endpoint receives certain *parameters* and returns certain *values*. Semantically, many of these APIs are object oriented: many endpoints accept some kind of ID parameter identifying a particular object to act on. In particular there are often endpoints for creating and deleting objects, and for reading or updating (some or all of) the properties of a particular object. The values of some object properties might be identifiers of other objects. Other endpoints might accept IDs of objects and collections and add or remove those objects to or from the collections. However, the semantics of these operations are not exposed by the endpoints; instead, they can be found in the (human readable) documentation. Anyone wishing to make use of the website's data must read that documentation to learn which endpoints manipulate which of the various site objects.

Our first objective was to eliminate the need for this human labor by creating a machine readable description that would permit *automatic generation* of a library that presented a website's data as a collection of (typed) objects in the local environment, such that reading or modifying the object's properties would automatically trigger the invocation of appropriate API endpoints to perform the requested modifications on the website. To do so, we needed to determine how *general* such a description would need to be, in order to capture the diversity of present-day web APIs.

5.1 Surveying Web APIs

We began our design of WoOPI by analyzing a *random sample* of web APIs to understand what was common among them. We sampled from ProgrammableWeb [31], a catalog of over 24,000 APIs, and designed WoOPI to be able to describe them. After completing our design, we *validated* it by choosing a *different* random sample of 60 additional APIs and assessing whether WoOPI could describe them. The ProgrammableWeb divides its APIs into categories. There are 505 categories—too many for our sample to cover completely. Our sample hit 10% of all categories including the 9 most popular categories: Mapping, Social, Search, Travel, Weather, Music, eCommerce, Financial, and Photos.

In our initial random sample of 40 web APIs we found the following commonalities: All of these APIs offer objects of various types, all identified by ID. They have one API endpoint for each type that accepts an object ID and returns that object's "basic" properties—generally those that are strings, integers, or other small data types. They provide other endpoints for accessing each "large" property of an object—such as an image or video file or an array of associated objects. Of the 21/40 APIs that allow users to create new objects, each provides a single API endpoint for doing so per type. Of the 22/40 APIs that allow users to delete objects, each provides one API endpoint per object that deletes an object by its ID. And of the 22/40 APIs that allow users to update objects' properties, 86% of them provide one API endpoint per object to update multiple properties, while the rest provide multiple endpoints to update different properties.

In addition to endpoints for manipulating single objects, many web APIs offer other functionality. Of the 36/40 APIs that provide more general methods, 34 (94%) support search only while 6% support other methods (e.g. an API endpoint that creates a unique URL or an endpoint that parses a message). For the APIs that provide a search method, 97% allow users to search by keywords. Of those, 83% of them return only a subset of each objects' properties through their search endpoint, while the others return all object properties.

5.2 Designing WoOPI to Expose Objects

We designed WoOPI to support APIs fitting the common characteristics we identified from our initial API sample. An example is shown in Figure 1. WoOPI is built on top of HAAPI [1], a lower-level ontology for describing API endpoints. HAAPI describes the endpoints, what parameters each one takes, and the properties of the object(s) it returns. This description was sufficient for HAAPI's intended usage—providing a general-purpose object *search* that returns a flat table of values—but is not sufficient to support object linking, navigation, and manipulation. WoOPI extends HAAPI with the information needed to describe and implement a local object model:

- the various *types* of objects returned by the API, and the properties of each
- for each endpoint, which input parameters are IDs for objects of which types
- for each endpoint, which properties of the returned values are IDs for objects of which types
- which endpoints can be used to read or update which properties of an object by ID, and which parameter identifies the object being retrieved or updated.
- which endpoints create or delete objects of which types
- which endpoints add or remove objects of which types from collections (those collections may themselves specified by IDs, or may be particular properties of other objects specified by IDs)
- methods that can be invoked on objects of a given type, and the endpoints that implement those methods (such as liking a video)
- other methods that can be applied to the entire site—in particular a search method that returns objects—and the endpoints that implement them

Figure 2 shows HAAPI and WoOPI descriptions of the Dailymotion API. HAAPI (right) describes the API endpoint's URL, parameters, the schema of response data, pagination, authentication and headers if any. Specifically, Figure 2 shows the HAAPI description of one API endpoint that is used for the site's method "search for videos'.' WoOPI (left) maps the API's methods and parameters (as described by HAAPI via the references "haapiDescription") to a "normalized" API of standard Schema.org types. Specifically, WoOPI describes the API's *objects* of different types (e.g. MusicPlaylist and VideoObject)) that can be read and updated, and API's *methods* (e.g. search videos). For each object type, WoOPI describes how they can be created, deleted, read and updated.

5.3 Designing WoOPI to Transform to Standard Types

Another goal of WoOPI is to support *standardization* of objects over multiple APIs. Since APIs currently follow no uniform convention for choosing property names for the website objects, WoOPI also provides for describing how to map property names in objects returned by the API to different, preferred property names. We use this renaming to support mapping API data into the Schema.org standard as we discuss below. Thus, if a websites returns video objects that have a `title` property, WoOPI can assert that this property corresponds to the `name` property of videos from the Schema.org ontology. The WoOPI description in Figure 2 shows the mapping between the playlist `fields` returned by the Dailymotion API endpoint `/playlist/{id}` and the `properties` of the `Schema.org/MusicPlaylist` type (e.g. `{"schemaProperty":"creator", "apiField":"owner"}`).

5.4 WoOPI Evaluation

To verify that WoOPI does support most web APIs, we picked another random sample of 60 APIs from ProgrammableWeb. 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 (described below) can be generated to query them and manipulate their objects. Of the 10% not supported, one requires two IDs to get an object, two require two API calls to get all basic properties of a single object, one requires both an ID and additional parameters to delete an object, and two require multiple setters to update object properties. It would be trivial to extend WoOPI to support APIs like these, but we leave this as future work.

6 SHAPIRJS: SHAPIR JAVASCRIPT LIBRARY

ShapirJS is a JavaScript library that uses the WoOPI description to wrap an API's data as typed objects in the local programming environment. Library users can read and write properties of those objects, and invoke methods on them or on the site, as if they were local, and the library takes care of making the necessary API calls to provide or modify the relevant data. Optionally, the WoOPI description can map objects to standard Schema.org data types. Doing so permits applications written using ShapirJS over one website to work, unchanged, over any other website providing semantically-equivalent data. In this section, we describe the design, implementation, and user-study evaluation of ShapirJS.

6.1 ShapirJS Design

6.1.1 Local Connected Objects. Unlike accessing APIs directly or through any other machine generated library from a low level API descriptions (e.g OpenAPI [39]), ShapirJS presents object connections implicitly without requiring the user to invoke API endpoints to traverse the connections. ShapirJS creates local proxy objects for each remote object, and translates property accesses into API invocations as needed. An example is shown in Listing 3. The user directly accesses a specific playlist in Vimeo using `let playlist = await vimeo.MusicPlaylist("ID")` function. They then directly access the playlist videos `let videos = await playlist.video` and the information in each video in the playlist (e.g. the comments of the first video as `videos[0].comment`). ShapirJS maintains the ID of the playlist internally so it can pass it to relevant API endpoints to read that playlist's `VideoObject` objects, then internally tracks the ID of each video object so it can use an endpoint to fetch the video `Comment` objects. This is made possible by the WoOPI description specifying the ID-valued properties that connect these objects.

Web sites often design their APIs to optimize performance for common cases. Their main object-reading methods may return only certain "basic" properties of objects, with other methods available for fetching other properties. ShapirJS hides this performance optimization complexity from the user. ShapirJS is lazy, and only

WoOPI

```
{
    "dailymotion": {
        "types": {
            "MusicPlaylist": {  // playlist  object
                "id": "playlistId",
                "read":{
                    "haapiDescription": "Dailymotion Playlist"
                },
                "create": {
                    "haapiDescription": "Dailymotion Create Playlist"
                },
                "delete": {
                    "haapiDescription": "Dailymotion Delete Playlist"
                },
                "update": [{
                    "haapiDescription": "Dailymotion Update Playlist",
                    "schemaProperty": "name"
                    },{
                    "haapiDescription": "Dailymotion Update Playlist",
                    "schemaProperty": "description"
                    },{•••},{•••}
                ],
                "properties": [{
                     "schemaProperty": "creator",
                     "apiField": "owner"
                    },{
                     "schemaProperty": "dateCreated",
                     "apiField": "created_time"
                    },{
                     "schemaProperty": "video",
                     "haapiDescription": "Dailymotion Playlist Videos",
                     "resultType": "[VideoObject]"
                    },{•••},{•••}
                ]
            },
            "VideoObject": {•••}, // video object
            "Person": {•••}  // person  object
        },
        "siteMethods": [{
            "type": "SearchAction",
            "haapiDescription": "Dailymotion Videos Search",
            "searchTerm" : "search",
            "object" : "VideoObject",
            "id": "videoId"
        }{•••},{•••}]
    }
}
```

HAAPI

```
{
    "description": "Dailymotion Videos Search",
    "url": "https://api.dailymotion.com/playlists",
    "parameters":[{
        "name":"search",
        "type": "string",
        "defaultValue": "Taylor Swift",
        "displayedName":"Search Playlists",
        "description":"Retrieve a list of playlists",
        "required": true,
        "display": true},
        {•••},{•••}],
    "responses":[{
        "name": "title",
        "displayedName": "Video Title",
        "description": "The title of the video.",
        "path": "list.title"},
        {•••}, {•••}],
    "pagination":{
        "type": "index",
        "typeParameter": "page",
        "resultsPerPage": 20,
        "resultsPerPageParameter": "limit"
    },
    "authentication":{
        "type": "apiKey",
        "in": "query"
    },
    "headers":[{•••}]
}
```

Figure 2: Part of a WoOPI description (left) that builds on a HAAPI description (right) for the Dailymotion website. HAAPI provides the API endpoint's URL, parameters, the schema of response data, pagination, authentication and headers if any (we are showing one API endpoint for searching Dailymotion videos). WoOPI maps an API of any methods and parameters (as described by HAAPI) to a "normalized" API of typed objects with create/read/update/delete methods. Specifically, WoOPI describes the site's types (MusicPlaylist, VideoObject, etc) and methods (search videos, search playlists, etc). Each type description includes read and write methods for the object's properties, and create and delete methods for objects of that type. The descriptions of the different API endpoints that WoOPI uses are fetched from HAAPI (highlighted "haapiDescription"). WoOPI connects different API endpoints through the objects' properties (e.g. playlist's videos are connected to the playlist through the "video" property)

"triggers" API calls for missing properties when the user actually tries to access them. For example, `vimeo.MusicPlaylist(<ID>)`, will only access the endpoint that returns the basic properties of the playlist, and will not invoke other API endpoints until the user actually uses them (e.g. `playlist.video` will call another API endpoint that returns the playlist's videos).

The fact that accessing certain properties requires triggering new API calls means that these accesses may be *asynchronous*. The JavaScript `await` syntax makes it easy to incorporate asynchronous

```
// 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:200});
// 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

actions into code; code prefixed by an `await` keyword (as shown in Listing 3) will yield control flow and resume when the awaited code completes. A ShapirJS user will need to check the (automatically generated) documentation of a given library in order to know which properties of an object require the await keyword. This abstraction leakage is unavoidable if we are restricted to providing a runtime-interpreted library; a WoOPI aware *compiler* could generate `await`s where necessary.

6.1.2 Documentation. In addition to providing the ShapirJS library, Shapir also uses the WoOPI description to automatically generate documentation of the local object model provided by ShapirJS. The documentation lists all object types and their properties (and methods as described below) declared in the WoOPI description. It also includes code snippets showing how to create and delete objects, access their properties and invoke their methods, and modify collections, including any necessary `await` keywords. Figure 8 (left) shows a code snippet generated by the ShapirJS documentation for the Songkick API.

6.1.3 Standard Types, Properties and Collections. WoOPI can describe an API using standard Schema.org types; ShapirJS then rewrites returned property names to provide those standard types in the local object model (and its automatically generated documentation). This allows developers to use the same code to access several different APIs that offer the same types of data. For example, the user can access and manipulate videos and playlists with standard Schema.org properties and methods regardless of whether these videos reside on YouTube, Dailymotion, or Vimeo. 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 that 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. This code would work unchanged given WoOPI descriptions of YouTube or Vimeo.

6.1.4 Search and Other Site Methods. In addition to objects with updatable properties, ShapirJS supports invocation of API-provided methods on objects as well as "site-wide" methods such as search. API endpoints that accept a particular type of object are wrapped by ShapirJS as methods for that object type in the local model. "Site" API endpoints become (static) methods of the object representing that site in the local model. Listing 3 shows a user invoking `vimeo.search()` as a method on the Vimeo web site; this method accepts a search parameter ("Adele") and returns `VideoObjects`.

This example demonstrates both the power and the current limits of standardization through Schema.org. If the user instead wished to search YouTube, they would simply replace it with `youtube.search()` and receive `VideoObjects` from YouTube instead. This works because Schema.org standardizes the notion of a search parameter for a `SearchAction`. However, YouTube would ignore the `sort` and `filter` parameters because these are specific to the Vimeo API and are not understood by the YouTube API (which uses an `order` parameter instead).

Ideally, we would also like to standardize the parameters for methods that are doing the same work. For example, if YouTube uses `order` to order the returned data while Dailymotion uses `sort`, it would be desirable to consolidate these two parameters into one (e.g. `sortBy`), just as we mapped the data properties returned by these APIs to standard Schema.org common properties. Unfortunately, Schema.org does not at present offer space in its ontology for such API parameters. Schema.org is designed to allow web sites' owners to describe the data on their web pages, and it does not provide a way to describe the parameters that can be used to query the data on those web pages. So, we did not have a common vocabulary to

use to standardize the API parameters. This would be a powerful future extension for Schema.org. WoOPI could already be used to map to standard parameters, but lacks the reach of Schema.org to advertise them as standard.

In the meantime, aligning to Schema.org *does* provide enough information to let us standardize property names and their manipulation through get and set methods, as well as collections with add and remove methods.

6.1.5 Implementing Search. ShapirJS hides significant complexity from the user around the way web site APIs implement search. Search APIs generally return "truncated" objects that contain only the most commonly accessed parts of the object, expected programmers to access more extensive "details" about each object using other API calls. ShapirJS abstracts this complexity away: invoking the search method provides a collection of objects, and ShapirJS seamlessly makes further API calls if and only if the user's code accesses properties that require those details.

For example, when the user searches Yelp for businesses, they would expect to read a collection of `business` objects with all the details about them. But the Yelp search API, like many others, returns a collection of "partial" `business` objects and requires further API calls to `/businesses/id` for each object to fetch its full details. From the sample of APIs that we have analyzed, we found that 16% of APIs follow this practice. ShapirJS deals with this implicitly without any intervention from the user. A ShapirJS search will return a collection of `business` proxy objects that appear to have all of their properties, invoking additional API calls when necessary to fill in properties the user chooses to inspect.

Search APIs also generally *paginate* their results, requiring endpoint invocations to specify specific pages or ranges for the search. ShapirJS takes care of all this for the user based on the description of the API's pagination behavior (which is inherited from the underlying HAAPI ontology). The user may optionally specify a number of results to override the default HAAPI value. ShapirJS uses this information and provides a common parameter called `numberOfItems`, from Schema.org, that can be used by users to specify the number of results (e.g. `yelp.search("Seafood", {"location": "NYC", "numberOfItems": 50})`).

ShapirJS thus permits the programmer to ignore performance considerations and simply access objects and properties as desired. But this also means it hides these performance considerations from the programmer, who may end up writing inefficient code as a result. We believe this trade-off of simplicity for performance is worthwhile when writing simple applications.

6.1.6 Authentication. In addition to pagination, ShapirJS handles authentication. The vast majority of APIs authenticate using an API key. They vary in how that key is delivered (e.g. as a query parameter or a header parameter), but the delivery mechanism is part of the underlying HAAPI description. Thus, all a ShapirJS user needs to do is specify the key. To support this, ShapirJS provides a common parameter called `apiKey` that allows the developer to specify their key/token; ShapirJS will pass it to the API in the appropriate way. ShapirJS provides an `init()` function for each API that can be used to pass the `apiKey` and its value to the site's functions (e.g. youtube.init({"apiKey": "<KEY>"})). For OAuth, ShapirJS uses the OAuth information provided by HAAPI to allow users to login to their accounts and give permission to the application to access the web site data through the API.

6.1.7 Error Handling and Debugging. ShapirJS tries to recover from minor errors (e.g. it will omit unknown query parameters). For more signficant errors, such as methods/types that are not supported by the API or API errors, it throws errors as feedback. ShapirUI's automatic documentation also helps avoid errors. Debugging is critical. We are working on progressively revealing the behind-the-scenes of each API call to the user for debugging. As we will mention in the next section (6.2.4), two study participants asked for this.

6.2 ShapirJS Evaluation

We conducted a user study between 16 subjects, comparing ShapirJS to Swagger Client [39], a widely adopted library for accessing web APIs. The study objective was to evaluate the usability and efficiency of ShapirJS focusing on the following question: Can programmers access APIs using ShapirJS faster and more easily than using the Swagger Client library?

6.2.1 Swagger Client Library. Swagger Client is a popular library for simplifying access to web APIs. Swagger reads an OpenAPI specification of an API and provides JavaScript methods for accessing the described API endpoints. OpenAPI specifies a unique string called `operationId` that is used to identify the API's operations (endpoints/methods). For example, the OpenAPI specification for YouTube API endpoints `GET /videos/video_id` and `DELETE /videos/video_id` might have the `operationId` "get_videos" and "delete_videos" respectively. Swagger Client allows users to use these operation IDs to access these API endpoints. Alternatively, users can use the `path` and the `method` of API endpoints. In addition to the `operationId` or the `path+method`, users need to specify the parameters, request body, and the authorization process for each API endpoint. Listing 4 shows a code snippet of ShapirJS (top) compared to the Swagger Client (bottom) for reading a playlist and its videos and searching for videos on Vimeo. Like ShapirJS, Swagger includes tools to automatically generate documentation from an OpenAPI description, that assists users coding to the Swagger Client library.

ShapirJS uses WoOPI, which extends OpenAPI, so comparing to Swagger Client is a natural way to assess the benefit of the extra semantics that WoOPI adds to OpenAPI.

An alternative "control" would be to have programmers code the traditional way, by manually authoring "raw" URLs targeting the API endpoints, and manually configuring authentication and other headers in the HTTP requests. But this would be a weak straw man. The widespread adoption of Swagger Client reflects the benefits of having a library automatically assemble those URLs from higher-level method invocations. Indeed, in a similar user study we carried out on ScrAPIr [1], developers on average required 27 minutes to write URLs accessing one API endpoint. In our user study, participants accessed 7 different API endpoints in one hour, so raw coding was infeasible as a control.

6.2.2 Procedure. We recruited 12 participants (7 female, 5 male, ages 19 to 37) for a one-hour user study. Participants' self-assessed

```
/****************************************** ShapirJS ******************************************/
// Get the playlist
let playlist = await vimeo.MusicPlaylist("8274189");
// Read playlist information
console.log(playlist.name, playlist.description);
// Get the playlist videos
let playlistVideos = await playlist.video;
// Search Vimeo
let videos = await vimeo.search("Adele", {sort:"relevant", filter:"trending", numberOfItems:200});


/************************************* Swagger Client *************************************/
const client = await SwaggerClient({
    url: 'https://api.apis.guru/v2/specs/vimeo.com/3.4/openapi.json',
    authorizations: {oauth2: {token: {access_token: '<TOKEN>'}}}
});
// Get the playlist
const playlistResult = client.execute({
  operationId: "get_album",
  parameters: {album_id:"8274189", user_id:"108506131"}
});
// Get the playlist's videos
const playlistVideosResult = client.execute({
  operationId: "get_album_videos",
  parameters: {album_id:"8274189", user_id:"108506131"}
});
// Search Vimeo
const videosResult = client.execute({
  operationId: "search_videos",
  parameters: {query:"Adele", sort:"relevant", filter:"trending"}
});
// synchronize all issued requests at this point
const results = await Promise.all([playlistResult, playlistVideosResult, videosResult]);
let [playlist, playlistVideosResponse, videosResponse] = results.map(result => result.body);

// Read playlist information
console.log(playlist.name, playlist.description);
// Read playlist videos information
let playlistVideos = playlistVideosResponse.data;
// Read searched videos
let videos = videosResponse.data;
```

Figure 4: An example of retrieving a playlist with its videos and searching for videos using the Vimeo API with both ShapirJS (top) and Swagger Client (bottom). Unlike ShapirJS, Swagger Client does not support automatic pagination of results, and leaves it to the programmer to figure out the type of pagination and write code that implements it. ShapirJS provides a property `numberOfItems` that lets users specify the number of results without worrying about implementing the different types of pagination that different APIs support. (We are not showing here how to go through multiple pages in Swagger Client to retrieve 200 items)

programming skills ranged from intermediate to advanced: 2 intermediate, 5 skilled, and 5 advanced programmers. Their experience with APIs varied: 5 had used them a couple of times, 5 had used them quite often, and 2 had used them all the time as part of their work/research.

Because we conducted a between-subjects user study, half of our participants were assigned to use ShapirJS, and the other half were assigned to use the Swagger Client. We used the *stratified randomization* method [25], where we grouped participants by expertise, then randomly partitioned each group so we get same amount of expertise on each condition. Participants were allowed to use the (automatically generated) documentation of the given API, Swagger Client and ShapirJS, and to look up solutions to issues they were facing during the study.

Participants, using either ShapirJS or Swagger Client, were assigned to write code to perform identical tasks using the Vimeo API. The tasks were the following:

(1) Get a specific playlist (album) from Vimeo (playlist ID and user ID were given)
   (a) Get the name and the description of that playlist
   (b) Get that playlist's videos
   (c) Get the comments of the 8th video in that playlist
(2) Search Vimeo for "Adele" videos and sort them by relevance
(3) Create a new playlist with the name and description "New"
(4) Update the description of that playlist to "Still New"
(5) Remove that playlist

Traditionally, performing these tasks would require programmers to assemble 7 HTTP requests: (1.a) `GET /albums/album_id` to get the name and description of the playlist/album. (1.b) `GET /albums/album_id/videos`, to get the playlist videos. (1.c) `GET /videos/video_id/comments` endpoint to get the comments for the 8th video in the playlist. (2) `GET /videos?query=Adele&sort=relevant` To search Vimeo for videos. (3) `POST /albums` with the appropriate parameters, to create a new playlist. (4) `PATCH /albums/album_id` with the appropriate parameters, to update the description of that playlist. (5) `DELETE /albums/album_id`, to delete that playlist.

In addition to assembling all these HTTP requests, the user would also have to configure authentication to the API. The Vimeo API uses the OAuth2 standard.

Instead of writing raw URLs, our subjects used the JavaScript interfaces provided by ShapirJS or Swagger Client. Listing 4 shows how users can perform tasks 1 and 2 using the Swagger Client and ShapirJS. Listing 3 further shows how users can perform all the tasks in 7 lines of code using ShapirJS; the Swagger Client code would occupy several pages.

Participants were allowed to use the documentation of the Swagger Client library and ShapirJS. ShapirJS documentation automatically generates code snippets of the API objects and site methods from WoOPI descriptions of individual APIs. Table 8 (left) shows part of the generated ShapirJS documentation for Songkick API.

6.2.3 Results. All participants who were assigned to ShapirJS (6/12) completed their tasks, with an average time of 8 minutes, and 5 out of 6 participants who used Swagger Client completed their tasks, with an average time of 45 minutes. ShapirJS was 5.6x faster than the Swagger Client for completing the same set of tasks. Figure 5 shows the task completion time for individual participants using ShapirJS (P1-P6) and Swagger Client (P7-P12). P12 (Swagger condition) did not finish, completing only 4/7 assigned tasks within the given time, despite rating their programming as advanced.

We conducted an unpaired t-test (after pruning the time for P12, who did not finish the tasks, to the full duration of the experiment (60 minutes)) to determine if there is a significant difference between the means of the two groups of participants. There was a significant difference in the completion times for Swagger Client (M=47.64, SD=6.26) and using ShapirJS (M=8.28, SD=3.65) with $p<0.0001$.

In an after study survey, we asked participants to rate how difficult it was to use Swagger Client and ShapirJS. 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 4 for Swagger Client and 1.3 for ShapirJS.

6.2.4 Participant Feedback. We asked participants about their experience accessing APIs using Swagger Client and ShapirJS.

Pros and cons of using Swagger Client. We asked participants what they did and did not like about using the Swagger Client library. They mainly liked that they can use functions (operationId) instead of assembling HTTP request to access API endpoints. One participant said "*the operation ID names were very useful; I liked having a concise name for each action. I like that it avoids using the URLs of the APIs.*" Participants also liked that Swagger Client can check the OpenAPI specification for required parameters and let the user know if they are missing these parameters "*nice that it could validate missing required parameters before sending the API request*".

Regarding dislikes, all participants found it very challenging to connect multiple API endpoints (e.g. playlist's videos), given that Swagger Client depends on OpenAPI which describes endpoints independently. One participant said "*It felt like it took a lot of writing to use the Swagger Client, especially when it came to performing multiple calls/chaining the results of calls together.*" And another said "had to figure out whether associated resources (e.g. comments on a video) were embedded in data I already had, or if I needed to make a new request". ShapirJS, unlike Swagger Client, automatically chains multiple API endpoints. Listing 4 (top) shows how ShapirJS smoothly handles connecting multiple API endpoints in `playlist.video` which returns the videos of a given playlist. Swagger Client (Listing 4 (bottom)) requires users to make individual requests for each API endpoint.

Many participants felt that having to check the OpenAPI specification, Swagger Client documentation and the raw API documentation (because sometimes the swagger documentation was not clear) was very daunting "*The Swagger Client syntax was different from the Vimeo API syntax which made them hard to integrate.*" In addition, the syntax of the Swagger Client was challenging for some participants "*Wasn't clear what arguments could be passed in and what their types were; e.g. how to pass in authorization information or how to set a request body.*" Finally, participants found the Swagger Client documentation to be poor "*The documentation for Swagger was very unclear, which made applying things from the Vimeo API like authorization or request body changes quite difficult.*"

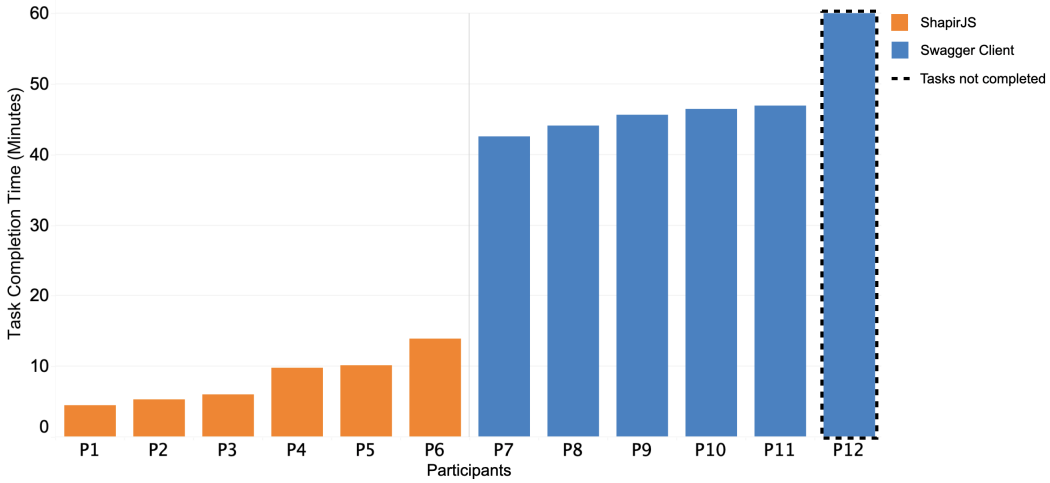


Figure 5: Task completion time using ShapirJS and Swagger Client.

Some of the just-described documentation problems might be ameliorated by improving the automatic generation of documentation from OpenAPI, perhaps using some of the ideas from our WoOPI documentation generator such as included code snippets for basic access tasks. Indeed, Swagger is well-known for their automatic generation of API documentation from OpenAPI through their Swagger Editor [40]. The Swagger Editor documentation is helpful to test the API and to know the paths and methods for the different API endpoints, but for users to use the Swagger Client library, they still have to go through the Swagger Client documentation which is of lesser quality.

However, even if the documentation were improved, users would still be forced to deal with some of the usability challenges inherent in the limited description provided by OpenAPI: for example, they would still need to do their own chaining of methods and would still need to manage authentication and pagination explicitly.

Pros and cons of using ShapirJS. We also asked participants what they liked and did not like about ShapirJS. One common answer for what they liked was performing complex tasks in simple syntax and in an object oriented style, where one said "*I like the fact that I can call complex functions like update, delete, and search in an Object Oriented style which means that I don't have to worry about different APIs format and stuff.*" And another said "*The use of getters (even with await) was much easier than constructing requests manually. The object-oriented interface design makes it extremely easy to know where to go to find information. I especially like how whenever possible, you call methods on named object types (e.g.* `MusicPlaylist.create` *or* `MusicPlaylist()`*) rather than top-level functions (which would be like* `vimeo.createPlaylist`*), because most times in plain JavaScript we do not actually get to see the types of objects, and this makes the types explicit.*" They also liked how ShapirJS standardizes access to APIs "*I really enjoy the literal approach to accessing and updating data. I think it's super cool that Shapir creates a shared vocabulary for various APIs. You can now learn the relationships between data once and apply that knowledge in multiple places.*" Finally, participants liked how well documented ShapirJS is, where one said "*It was very easy to use because it's well-documented. I liked that the function names were direct and to the point.*" Figure 8 shows part of the code snippet generated by the ShapirJS (left) documentation for the Songkick API. Regarding the dislikes, participants were confused about Schema.org's use of singular noun forms to name multi-valued properties "*The wording of some of the properties like how (video) refers to a list of (videos).*" Schema.org limitations will be discussed in detail in the discussion section (10.1). Finally, one participant preferred to know the number of API calls behind the ShapirJS functions "*One thing that could be challenging is being able to tell how many API calls are being made - since many APIs are charged by the number of API calls*"

Improvements to the ShapirJS syntax. We asked participants if they would prefer a different syntax for the ShapirJS functions that they used to perform the tasks. Two participants suggested a different syntax for search: One was to use key-value pairs for all the parameter including the search term `vimeo.search({''query'': ''Adele'', ''sort'': ''relevant''})`, which we already support. And the other was `vimeo.search(''Taylor Swift'').sort(''relevant'')`. We decided to make the search term a positional parameter because most search APIs (97% based on our API analysis) use that parameter. For the second suggestion, we actually considered this, but this syntax can get complicated if the search API has a lot of parameters. One participant also suggested a different syntax for create. Instead of using `vimeo.create.MusicPlaylist()`,they suggested `vimeo.create({''object'':''MusicPlaylist'',...})`. This suggestion seems reasonable and consistent with the other functions (`delete()` and `search()`). One participant suggested a transaction model, where setters modify local data and another function is invoked to push these changes to the API "*make the setters only affect the local value, have the playlist object internally queue the changes that have been made, and then call something like (await playlist.save()) to commit the changes. That could potentially reduce the number of API calls as well, depending on the API.*" This might improve performance, but at the cost of simplicity, contrary to our current goals.

7 END-USER APPLICATION AUTHORING WITH SHAPIR AND MAVO

We integrated ShapirJS with Mavo, a language that empowers people to create interactive web applications by authoring HTML. Mavo assumes the data for such applications is in a single JSON file stored on a site like GitHub. We integrate with Shapir by leveraging ShapirJS to instead access data behind web APIs. Combined, ShapirJS and Mavo make it possible to create standalone web applications that manipulate data over multiple web APIs without writing any JavaScript or back-end code. As with ShapirJS, our goal is to permit the author to ignore the difference between data they are managing locally and data stored behind APIs.

Mavo-Shapir provides Mavo users with a high level syntax which does not require them to assemble URLs, and seamlessly takes care of authentication and pagination. Furthermore, it uses standardized Schema.org types to smooth out any differences in data from different sites and present them as Schema.org objects. Authors can write their HTML based on the Schema.org types they are working with, and changing data source becomes as easy as changing one `mv-source` attribute. In contrast, when working with APIs directly, changing data source often requires extensive code changes, or even a complete rewrite.

At present, Mavo-Shapir only supports *read* access to web APIs; providing write access is conceptually straightforward but will require substantial modifications to Mavo that we hope to undertake in the future.

In this section, we describe the language modifications we made to Mavo to describe connection to APIs, then describe a user study that demonstrates that authors can use these modifications to write applications that connect to APIs.

7.1 Mavo-Shapir Design

Mavo by itself allows users to author applications in HTML that read (and store) data at a data source specified by the `mv-source` (or the `mv-storage`) attribute; Mavo then renders the data and provides an editing interface for it based on the existing presentation. With Mavo-Shapir, we extend this functionality to permit Mavo to read data stored behind APIs described using WoOPI. Mavo-Shapir is implemented as a Mavo plugin that registers a new type of data source. Mavo authors invoke it using `mv-source="shapir"` on their Mavo root element. They can then point at an object with a specific id by specifying an `mv-source-id` attribute in their html, or access an entire collection of objects using the `mv-source-search` attribute. Parameters for the type of search to perform are provided via `mv-source-*` attributes.

For example, Figure 6 shows a Mavo application to search both Yelp and Foursquare and list their restaurants. In this application, instead of rendering data fetched from a static JSON file, Mavo-Shapir fetches the data by executing a search (as specified by ShapirJS) on Yelp and Foursquare. Given that the data schema is standardized across similar APIs, like Yelp and Foursquare, the user can use the same property names with these APIs (aggregateRating, priceRange, etc).

While the standalone Mavo library generally fetches and manipulates one file containing all the data, it is clearly not scalable for Mavo-Shapir to fetch *all* the data behind an API. Instead, Mavo-Shapir allows a user to access specific data items identified by their ID, or to access the collection of items returned by invoking the API's search operation. Mavo-Shapir translates HTML attributes such as `mv-source-search="Jacket"` to API parameters described in WoOPI before querying data. Users declare which data provider(s) they wish to query via an `mv-source-service` attribute. Users can use this HTML-based syntax to specify any other criteria supported by the API for more granular data querying. For example, they can filter products from Etsy by `mv-source-min_price="10"`, `mv-source-max_price="100"`, and sort them via `mv-source-sort_on="price"`. For each site described by WoOPI, Shapir provides documentation for the user that specifies the parameters supported by each site, and provides sample code that is dynamically generated.

At present, Mavo-Shapir provides only *read* access through web APIs. Mavo applications that fetch data from web sites can still manipulate and store that data in Mavo's usual storage locations. Thus, for example, an author could use Mavo-Shapir to connect to YouTube to search for and play videos, while managing those videos in playlists that they store locally. Enabling Mavo to push updates back to web sites requires changes to Mavo's implementation of its core storage model, which we hope to pursue in the future.

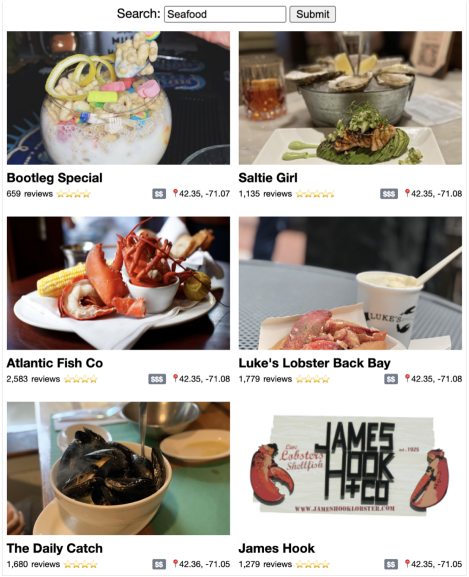
7.2 Mavo-Shapir Evaluation

We conducted a user study to evaluate the usability of the Mavo-Shapir plugin to answer the following question: *Can Mavo-Shapir reduce the effort and skill required to build applications that access data through web APIs?*

It has already been shown [43] that non-programmers with basic HTML knowledge can quickly create Mavo applications to manipulate data that is defined and stored locally. The improvement provided by Mavo-Shapir is to empower those same types of authors to build the same kinds of applications but interact with data behind web APIs. Therefore, our study design began with Mavo applications *already implemented* to interact with local data—the kind of applications we already know such authors can build—and investigated whether users could modify those applications to interact with web APIs. This reduced the need for users to familiarize themselves with all details of Mavo, as well as reducing the time they needed to spend on their tasks, while still studying the novel element.

We recruited 16 participants (9 female, 7 male, ages 18 to 60) for a one-hour user study. Of these, 8 identified as beginner or intermediate in HTML, and 8 as advanced or expert. Their programming skills ranged from none to skilled: 2 with no programming skills, 6 beginners, 6 intermediate and 2 skilled. Mavo has been deployed for several years, and has attracted a small user base that we included in our recruitment. In terms of Mavo familiarity, 7 participants had used Mavo before, 4 had heard of it but not used it, and 5 had never heard of it.

7.2.1 Procedure. Sessions were conducted one-on-one and were limited to one hour. We started the session by giving a quick overview of Mavo, focusing on the main functionalities of Mavo




```
<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 6: Search Yelp and Foursquare for businesses.

that the participants would need for the study. Then, we went over the (generated) Mavo-Shapir plugin documentation, and explained how it works with an example. We then assigned all the participants 4 apps/tasks. In the first three tasks, we asked participants to complete the functionality of three different applications, by making these applications read data from web APIs using the Mavo-Shapir syntax. Participants were given the HTML, CSS and (local data) Mavo markup, and only had to add Mavo-Shapir attributes to complete their functionality. As a greater challenge, the last task asked them to build a complete Mavo application from scratch, which also integrated data from multiple web sites. Each participant was assigned the following tasks listed in Figure 7. The highlighted code is what participants needed to add to perform the tasks in each one of the four Mavo applications.

Similar to the ShapirJS user study, participants were offered to use the generated Mavo-Shapir documentation for the API, which includes code snippets of basic Mavo applications for WoOPI descriptions of individual APIs. Table 8 (right) shows part of a Mavo code snippet for Songkick search API. These generated code snippets helped our participants build running Mavo applications (App#4) in several minutes. Our documentation does not generate code snippets for searching multiple APIs, but it describes how to do so.

7.2.2 Results. All participants were able to finish all of their tasks. The average time participants, programmers and non-programmers, spent to finish the first three tasks was 3 minutes, 1 minute and 45 seconds respectively. For the fourth task, where they were asked to build an entire application from scratch, the average time was 4 minutes. Table 1, shows a breakdown of the average time non-programmers and programmers spent on each of the four tasks/apps. On the first task, non-programmers spent a bit more time on average compared to programmers, but for all the other tasks, both programmers and non-programmers spent roughly the same amount of time on average. Especially for task 4, this surprisingly rapidly completion benefited from the availability of code snippets that could be copied from the (automatically generated) documentation.

Table 1: Average time (in minutes) spent on the 4 tasks/apps by non-programmers and programmers

	App#1	App#2	App#3	App#4
Non-programmers	04:00	01:15	00:40	04:00
Programmers	02:57	01:00	00:50	04:00

Following are some of the challenges that participants faced with the Mavo-Shapir syntax.

Retrieving information by search versus by ID. Four of sixteen participants did not understand the difference between searching a site by keywords and retrieving information about a specific object by its ID at first. This may be because users typically search web sites by keywords, and unless they look at the URL, they do not even realize that IDs exist. Typically, even someone searching for a specific playlist or video is likely to perform a suitable keyword search then click the item in the result list.

Debugging. One of the limitations of Mavo-Shapir is the lack of meaningful feedback for errors. For example, when participants forgot `mv-source="shapir"` or any of the `mv-source-*` attributes, only a generic error was shown, instead of a more specific message that would guide them to debug. Shapir interacts with APIs through ShapirJS, and could show error messages returned by the API. ScrAPIr, for example, shows human-readable error messages to their users while authoring the HAAPI description [1]. We can

App #1. Dailymotion

- Look up a playlist from Dailymotion by its ID.
- Make this application dynamic where the user can enter the id of any playlist and the application will show the playlist's videos

App #2. YouTube

- A fixed search for videos
- Make the search dynamic where the user can enter any keyword(s)
- Order the result by date
- Display the description of each video

App #3. Yelp

- A fixed search for businesses
- Make the search dynamic
- Show the price range for each business
- Show only 20 businesses
- Replace Yelp with Foursquare

App #4. A from-scratch event search app

- Build an application that searches SeatGeek for venues
- Show the events for each venue
- Add Ticketmaster and Songkick to SeatGeek to search for venues and their events
- Restrict the location of the venues to be the "US"

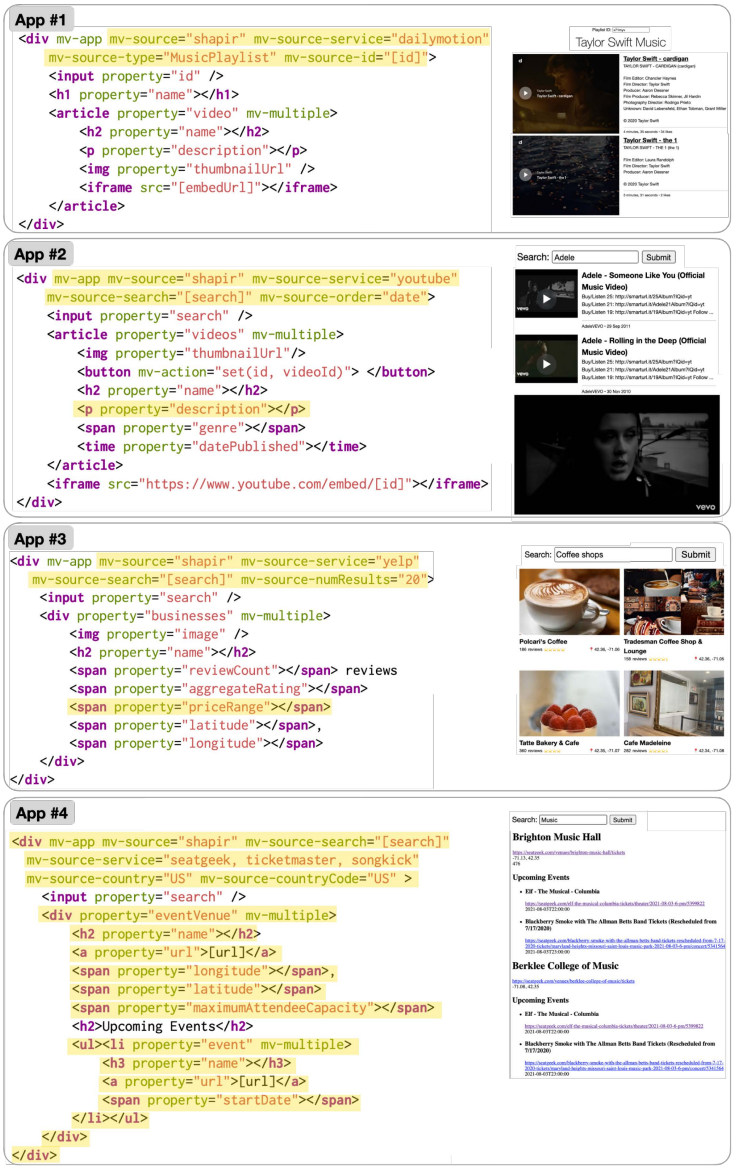


Figure 7: Mavo-Shapir user study tasks with their solutions. Participants were assigned 4 applications with tasks (left), and the highlighted code (right) is what participants added to perform the given tasks.

certainly use they same heuristics and show these messages to the Mavo users.

Query parameters when searching multiple sites. For reasons discussed in Section 6.1.4, Schema.org only specifies a single parameter—a query string. It does not offer any standardized names for the different parameters that could refine this search, such as ordering or filtering on certain attributes. Thus, Shapir standardizes the search term but not additional query parameters of search APIs. Users searching multiple sites users may therefore need to repeat the same query parameters with different names for different APIs. For example, for the last task in App #4 (Figure 7), participants needed to specify `mv-source-country="US"` for SeatGeek and `mv-source-countryCode="US"` for Ticketmaster. Nearly half of all participants (7/16) found that task challenging because of this. Namespacing query parameters that are specific to one site only (e.g. `mv-source-seatgeek-country="US"`) might help alleviate this issue in the future.

7.2.3 Participant Feedback. In a post study survey, we asked participants to rate how easy they found the Mavo-Shapir syntax with the four tasks assigned. Participants answered all questions with a

five-point Likert scale, from 1 (very easy to use) to 5 (very difficult to use). The average ratings were 1.75, 1.125, 1.125, and 1.5 for the four tasks respectively.

Participants found Mavo-Shapir to be easy to use and liked the documentation presented by Shapir. One participant said "*The documentation and examples were presented very clearly, it was nice to be able to go from static to dynamic site in just a few lines of edits.*". Participants also liked how Shapir standardizes the schema for sites that offer the same types of data "*Shows the versatility of Mavo-Shapir, being able to change only the name of the website and achieve different results*". Another said "*I liked how it showed Mavo+Shapir's uniformity across fields in API results and how it's possible to change the site it queries just by changing the mv-source-service from yelp to foursquare.*" Finally, participants were impressed by how quickly they can build applications that access multiple sites "*Neat to see how quickly an MVP site can be spun up that involved combining multiple data sources.*"

8 SHAPIRUI: A VISUAL AUTHORING TOOL FOR WOOPI

We have now demonstrated the benefits of using the WoOPI schema to describe an API so that ShapirJS and Mavo-Shapir can simplify access to it. In this section, we present ShapirUI, a simple GUI for authoring WoOPI descriptions and a user study showing that it works well.

8.1 ShapirUI Design

ShapirUI allows users to author WoOPI description and map their schemas to Schema.org vocabulary, as shown in Figure 9. It guides users to record key WoOPI information:

- what types of objects are provided by the API, and what are their properties?
- what are the standard names from Schema.org for these types and properties?
- what API endpoints are used to create, read, update, and delete each type, and what parameters are passed to them?

ShapirUI uses heuristics and auxiliary information from several sources to help reduce the effort of answering these questions.

8.1.1 The GUI. ShapirUI presents WoOPI information in two columns, as can be seen in Figure 9. On the left are types and properties (preferably drawn from Schema.org). On the right beside each type is the API endpoint for reading objects of the corresponding type, while beside each property is the field in the returned object that corresponds to that property of the type. Also on the left with a given type are other methods for the type (in purple boxes)—create, delete, and update, and any other special methods. Beside each on the right will be the API endpoint for invoking those methods. The UI offers typical affordances for adding types, adding properties and methods to a type, and adding endpoints and their fields corresponding to those methods and properties.

To begin, the user enters the URL of the web site they would like to describe. This can be the root URL or the site's API URL. ShapirUI then fetches the HAAPI description of all the site's API endpoints. If a HAAPI description does not exist, it can be authored using ScrAPIr's graphical authoring tool [1]). The endpoints of the HAAPI description determine the permitted values in the right column of ShapirUI—the endpoints that can correspond to the WoOPI types and create/read/update/delete methods being described (as shown in Figure 9).

8.1.2 Heuristics for Matching Types and Properties. ShapirUI then attempts to help the user identify relevant types from Schema.org that may be suitable for describing the API data. ShapirUI queries the Klazify API [26] which returns a set of *categories* for the web site such as Education, Banking, or Photo & Video Services. ShapirUI also extracts the HAAPI description of the API endpoints (each endpoint's English-language descriptions as well as the terms in the endpoint's URLs which may be English words). It treats the combination of category and HAAPI terms as a heuristic "description" of the web site.

ShapirUI then looks for Schema.org types whose English-language description (as found on the Schema.org site) overlaps with the extracted terms, and suggests those types to the user. ShapirUI calculates the *cosine similarity*, which is often used to measure document similarity in text analysis [15]) between the API's Klazify/HAAPI description terms and Schema.org types. It displays the top 5 similar types with the rest in the "Choose Type" dropdown provided for selecting additional suggested types as well as all the other types. The user can choose from these suggested types or start from scratch with a dropdown list of all the Schema.org types, and they can remove selected types.

For every selected Schema.org type, whether it was suggested by ShapirUI or manually selected by the user, the user needs to match it to the API endpoint that reads an object of that type. ShapirUI helps the user with this matching by calculating the cosine similarity between the selected Schema.org type description and all the API endpoints' descriptions and URLs (from HAAPI). It displays the most similar (by cosine) API endpoint next to the selected type (e.g. Figure 9 shows how the `ImageObject` type is assigned to the "Unsplash Photo" endpoint, from a list of API endpoints, based on highest cosine similarity). The user can change the selected API endpoint by selecting a different one from the API endpoints list.

Once the API endpoint for reading a (Schema.org) type is selected, ShapirUI helps the user match the API fields returned by the API endpoint to Schema.org properties of the type. ShapirUI also helps the user with this matching by calculating the cosine similarity between the description of each property on the Schema.org site and the selected API endpoint's *fields*, then shows these properties and most-similar fields next to each other. As shown in Figure 9, ShapirUI displays the `description` and `dateCreated` from `Schema.org/ImageObject` to the `description` and `created_at` fields from the API endpoint "Unsplash Photo". The user can then, manually match more API fields to the type's properties and can also remove the suggested ones. Some of the Schema.org properties can have complex values of another type (e.g. an `author` can be a `Text` a or a `Schema.org/Person`). When the user chooses one of these properties, ShapirUI will display a popover asking the user to choose the type of this property.

8.1.3 Example. If a user wants to describe Unsplash API collections of images using the ShapirUI, the user would first enter the Unsplash URL, as shown in Figure 9. The ShapirUI will then extract the web site's categories (Online Image Galleries, Online Communities,

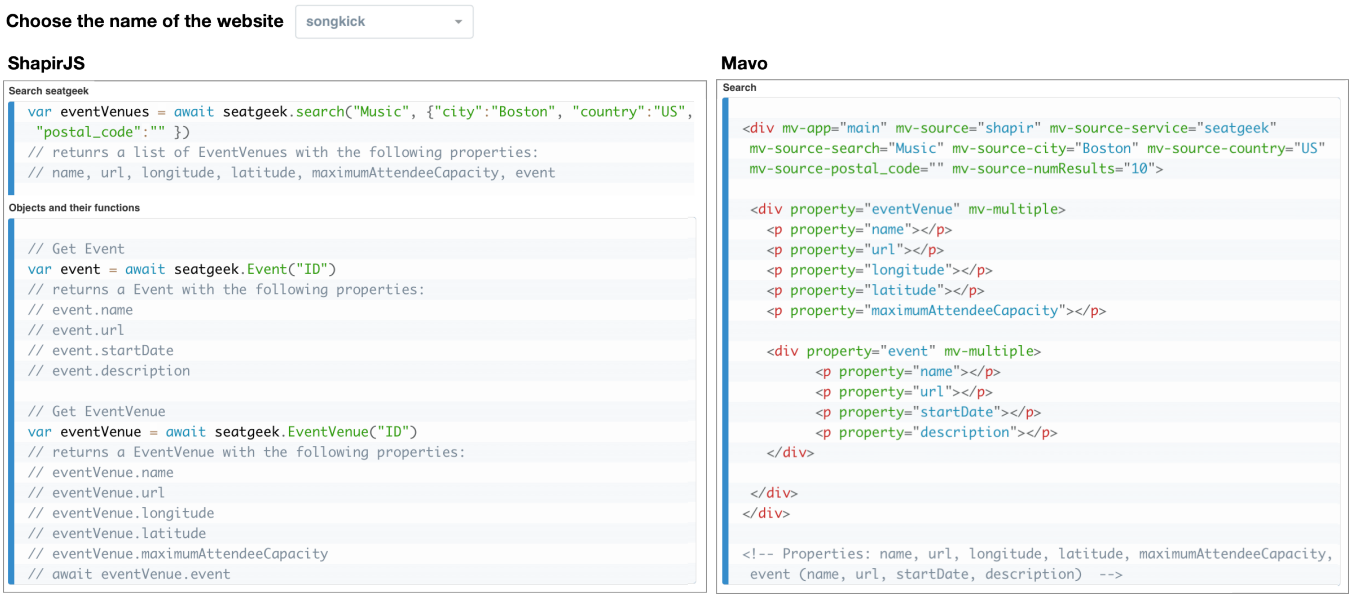


Figure 8: ShapirUI generates documentation in the form of code snippets for APIs that are described with WoOPI. This figure shows parts of the ShapirJS and Mavo code snippets for the Songkick API for events and venues. These code snippets describe the types of objects, their properties, and methods supported by the API.

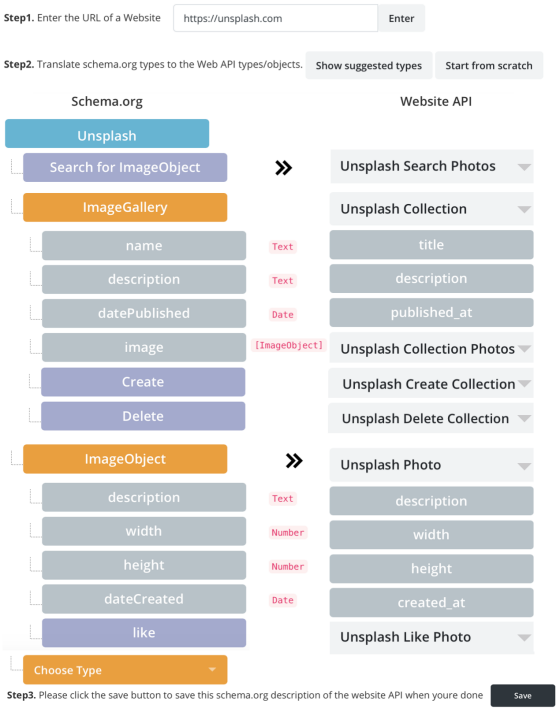


Figure 9: ShapirUI. (1) Enter the web site's URL. (2) Choose Schema.org types and properties to map the web site's API to Schema.org vocabulary. And describe the site and objects methods (3) Save the description to create a WoOPI for this API.

and Photo & Video Services), using the Klazify API. ShapirUI will ask the user to either choose to show suggested types or start from scratch. If the user chooses to show suggested types, ShapirUI will display two Schema.org types: `Schema.org/ImageGallery` and `Schema.org/ImageObject` (based on the cosine similarity calculated between Schema.org types and the site's categories and HAAPI descriptions). And for each type, ShapirUI will suggest an API endpoint that reads an object of that type (also based on the cosine similarity between that type and the HAAPI descriptions). For example, the `ImageObject` type will be matched with the `/photos/id` API endpoint. Some of these mappings between the types and API endpoints might not be correct—for example, ShapirUI will also suggest to match the `ImageGallery` to the `/photos/id` endpoint, instead of the `/collections/id` endpoint. The user can check these suggested mappings and choose the right API endpoint for each type. Then, for each matched API endpoint, ShapirUI will match some of the API endpoint fields to the type properties (those fields and properties that have a cosine similarity above 0). We choose to include all the properties and fields that are similar, even those with low similarities, because the user can easily remove these properties and add new ones. For example, ShapirUI will display the `description`, `width`, `height`, and `created_at` from the API endpoint "Unsplash Photo" (shown in Figure 9), and match them to the `ImageObject`'s properties `description`, `width`, `height`, and `dateCreated`. The user can then select the additional properties from the Schema.org types and map them to the appropriate API endpoint fields. To be able to read the collection *images* from Unsplash, the user needs to choose the property `image` from the `ImageGallery` type. According to Schema.org, the `image` property can be either an `ImageObject`, a

URL or of a primitive type. So, if the user selects the image property, ShapirUI will display a popover next to the image property asking the user to choose the type of the image and if it is either a collection or a single value. When the user chooses the type of image property to be an ImageObject, the image property will be added under the ImageGallery and a dropdown of API endpoints will be displayed next to the image property (as shown in Figure 9). The user then needs to choose the API endpoint that returns the collection's images /collections/id/photos. This image property will connect the ImageGallery type to the ImageObject type, allowing the user to read the collection (image gallery) images by invoking the image property.

8.1.4 Adding Methods. To describe each type's methods (create, delete, update, etc), the user can click on that Schema.org object/type. ShapirUI will display a sidebar asking the user to configure update, create, delete, or any additional methods (e.g. like) provided by the API (Figure 10(B)). For the create method, ShapirUI will simply ask the user to choose the API endpoint that creates an object. For the update and delete methods, ShapirUI will ask the user to specify endpoints as well as the object ID parameter. The properties that the user can update, or specify when creating new objects, will automatically be fetched from the HAAPI description. The user does not need to specify that in ShapirUI.

ShapirUI also allows the user to specify the search method, by clicking on the site node. ShapirUI will display a sidebar asking the user to pick the API endpoint, the search term parameter, the ID of the returned objects, and the type of data returned by the search API endpoint. Some APIs have one API endpoint that is used to search everything in the site, and other APIs provide multiple API endpoints that can search for different types of data. For example, Unsplash API has three search API endpoints: collections, photos, and users. ShapirJS allows users to add multiple search endpoints (shown as purple nodes in Figure 10(A)). For the search method, the user does not need to specify the search parameters (other than the search term). the ShapirUI will automatically fetch these parameters from the HAAPI description as well.

Once the user is done describing the objects and methods for the API, they can click on the save button, which will create the WoOPI description for that API. This will automatically generate the ShapirJS, client code and documentation for this API.

We are still completing the implementation of ShapirUI. At present it supports everything required for any WoOPI description except that: (1) ShapirUI does not yet support whole-site methods other than search; (2) it also does not support specifying the add and remove methods on collections of objects. We expect to have these features implemented soon.

8.2 ShapirUI Evaluation

We conducted a user study to evaluate the usability of the ShapirUI for making WoOPI descriptions aligned to Schema.org types. We focused on the following questions: (i) Can first-timers use ShapirUI to make a WoOPI description? (ii) How easy is it to understand the Schema.org vocabulary and choose the appropriate types and properties for a given API?

We recruited 12 participants (10 female, 2 male, ages 20 to 44) for a one-hour user study. Programmer skills ranged from none to advanced: 1 non-programmer, 4 beginners (understand concepts but frequently need documentation), 1 intermediate (does not usually need documentation), 3 skilled (can write complicated programs), and 3 advanced programmers (professional). Programmer experience with APIs varied: 1 had used them once, and 6 had used them a couple of times, 3 had used them quite often, and 3 had used them all the time as part of their work/research.

8.2.1 Procedure. Participants were assigned one of two APIs to describe using the ShapirUI : Dailymotion and Unsplash APIs. Half of our participants were randomly assigned to each. Participants were asked to make WoOPI descriptions detailed enough to let allows other users complete the following tasks: for Dailymotion, (1) Retrieve a specific playlist from Dailymotion with its name, description, creator, date created, and videos, and, for each video in it, get the name, description, URL, duration, thumbnail URL, and creator. (2) Search for videos; (3) Create a new playlist; (4) Update a playlist's properties; (5) Remove a playlist; (5) Add a video to a watch-later playlist; for Unsplash: (1) Retrieve a specific collection from Unsplash with its name, description, publish date, update date, and images, (2) for each image in the collection, to get its description, height, width, owner, thumbnail, and total number of likes on that image; (3) Search for images; (4) Search for collections of images; (5) Update a collection; (6) Remove a collection; (7) Like an image.

The study session began with a quick demo of the tool. After finishing their session, each participant answered a survey that asked them to reflect on the tool's usability and efficiency.

8.2.2 Results. All participants were able to finish their tasks with an average time of 11 minutes. The following are some of the challenges that participants faced usingShapirUI for the first time.

Web site versus object methods. With some of the methods, the distinction between the methods performed on objects versus the ones performed on the web site was not very clear at first to all participants. Some of the participants were confused about where to configure the search method versus where to configure all the other methods. Since search is a method that we perform on the web site rather than individual objects, ShapirUI requires users to click on the site node to integrate the search methods (as shown in Figure 10 (A)). The participant's confusion came from the fact that they search the site for a specific type, so naturally, they would want to click on that type to configure the search method. But some web sites offer only on search API endpoint that allows users to search the site for multiple types (e.g. YouTube search API). In this case, it would not appropriate to have the search to attach to specific types.

Object methods. ShapirUI allows users to configure, for every object, create, delete, update and other methods (Figure 10 (B)). Some participants did not understand at first what is the difference between (create, remove, and update) and other methods. One participant said "*The (Like a photo) task was challenging at first, because I thought in order to like you would click on (create) as if you're creating a like.*". We agree with the participants, that separating these methods might be confusing. We will try improve our UI to better guide the user to describe the different methods clearly.

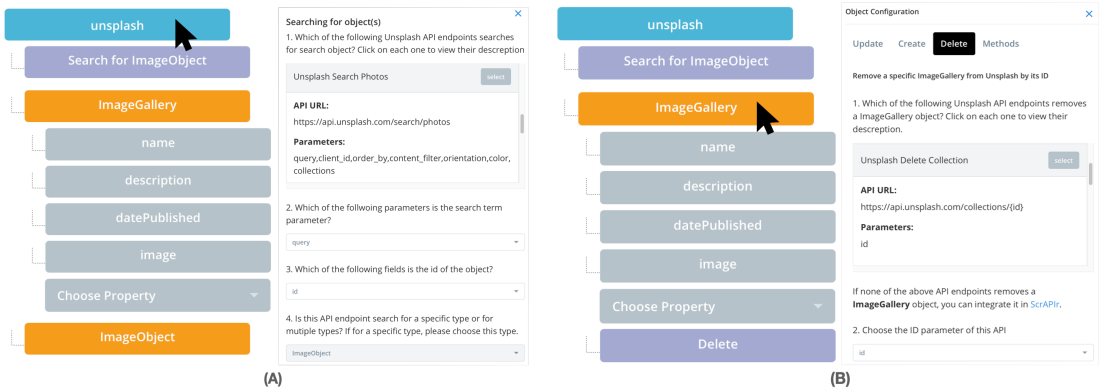


Figure 10: (A) Describing the "search" functionality of the API (B) Describing "create", "delete", "update" and other methods of the API

Schema.org types and properties. None of our participants were familiar with Schema.org vocabulary. The main challenge that they faced was understanding which Schema.org types and properties to use with the given API. For example, for Unsplash collection of images, participants were not sure if they should use Schema.org `ImageGallery`, `Collection` or `CollectionPage`. In addition, the `ImageGallery` type had a property `image` of type `ImageObject`. Participants expected a property `images`, not `image`, with the type "list" of `ImageObject`. All properties in Schema.org are allowed to have multiple values. But, the Schema.org documentation does not specify that in their types' pages. In addition, all complex-valued properties can also have Text or URL value, but Schema.org does not specify that very well in their types' pages.

Feedback. ShapirUI does not really give feedback to the user about whether their mapping between API objects and Schema.org types is correct or not. Mainly, because there is no way to check. What we are proposing in this paper is novel, and there is no repository of similar mappings on the web that we can check against.

8.2.3 Participant Feedback. In the survey, we asked participants to rate how usable and easy to learn they found ShapirUI. 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 for usability and learnability were 2 and 1.5 for the two API tasks respectively.

Participants found the UI to be intuitive and easy to use. One said "*The UI was very easy to understand, the objects were easy to map from Schema.org to the specific API objects, really cool to see the update/create/remove features built in to each object.*" And another said "*The display with color coding seems user-friendly and I liked how the matching worked. If I selected something from Schema.org and the matches from Dailymotion didn't make sense, then I knew something was wrong with my selection*". Another said "*I love the block interface as it clearly shows the hierarchies and matches between schema and the API. The search functionality and adding methods are quite intuitive once you know how to access them by clicking on the blocks.*"

9 IMPLEMENTATION

Shapir consists of (1) a front end web interface (ShapirUI), built using open Web technologies (JavaScript, HTML, and CSS), and (2) a back end component, which stores the WoOPI descriptions, built using the Firebase Realtime Database. ShapirJS reads a WoOPI description from Firebase and uses it to generate standard functions that can be used by users to access APIs. ShapirUI is a GUI that can be used to create the WoOPI descriptions. ShapirUI uses a cosine similarity algorithm to find relevant types and properties to the web site's API that the user is interested in accessing. Shapir integrates with Mavo through a JavaScript plugin that adds Shapir as a new Mavo backend. It takes care of translating HTML attributes to ShapirJS function calls that allow HTML authors to query APIs and retrieve their data.

10 DISCUSSION

Accessing data through web APIs has become essential to modern applications. But it is also a significant obstacle: the complexity and variability of these APIs means that even experienced programmers need to spend significant time working to access each, and that less-experienced and non-programmers might not be able to do so at all. In this work, we have demonstrated a collection of related tools that allow end users to treat data behind web APIs as local objects in their own computational environments, dramatically simplifying the authoring of simple applications for interacting with that data. In combination with Mavo, our tools allow an author to create complete web applications that access data behind these APIs, entirely in HTML without authoring a single line of JavaScript.

Key to our approach is the idea of *standardizing* web APIs at a *higher level of abstraction* than they currently support. We argue that instead of arbitrary functional endpoints accepting and returning primitive string and integer arguments, web APIs should be modeled as *methods*, particularly the canonical create, read, update, and delete quartet, and add and remove for collections, operating over *linked, typed objects* with type-specific *properties*. This is the dominant abstraction for programmers in their local environment, and extending it to web APIs can dramatically simplify the incorporation of those web APIs into applications.

We have developed WoOPI, a prototype standard schema for describing web APIs according to this abstraction. This schema is simple enough that even non-programmers can make WoOPI descriptions using the ShapirUI GUI. Given a WoOPI description, the ShapirJS library can expose the data behind the API as (proxy) objects with readable and writeable properties in the local environment, allowing a programmer to access the data without considering the API itself at all. Shapir also uses the WoOPI description to automatically generate documentation of the object model, along with code snippets the developer can copy. Finally, integrating Shapir with Mavo, which lets people author data management applications in HTML, allows people to create fully functional applications over web APIs without writing a single line of JavaScript.

Our work also demonstrates the benefits of mapping APIs to *standard* object types. Applications that are written to operate on those standard objects types will work, unchanged, on any web site whose WoOPI description exposes those standard object types. Our ShapirUI leverages assorted heuristics to help authors map an API to appropriate types from the Schema.org repository.

We advocate for *standards over platforms*. While many platforms are emerging that proxy numerous APIs through one standard meta-API, such approaches create unnecessary performance and control bottlenecks. In a standards-based future, each website could publish its own WoOPI description allowing any client program to interact with web site directly.

10.1 Limitations and Future Work

We chose Schema.org as our repository of standard types. Schema.org aims to shoehorn the entire web into these types, which makes their schema rather "sloppy". For example, many properties of many types can be either (i) an object of some other type, (ii) a *collection* of such objects, or (iii) a text string that somehow describes such objects. ShapirJS attempts to disambiguate, but can be fooled. A more precise specification would be useful.

Schema.org also is far weaker at describing standard *methods* than types. Although there is a canonical `SearchAction`, Schema.org only specifies a single parameter—a query string. It does not offer any standardized names for the different parameters that could refine this search, such as ordering or filtering on certain attributes. The lack of standard attributes means that users integrating multiple web sites' search endpoints will be unlikely to standardize the parameters. This means that (as we already saw in our video search example) authors will need to specify such parameters on a site-by-site basis, harming portability. There could be significant benefits to extending Schema.org to standardize methods as well as types.

One limitation of mapping web APIs' schemas to Schema.org, or any other standard schema for that matter, is that there is no way to validate this mapping (if these types/properties are the right ones for those APIs). One challenge with this is that different users might map different sites offering the same type of data to different types/properties. For example, `Schema.org/VideoObject` has `creator` and `author` properties. Both properties can be used to represent the owner of the video. Using the ShapirUI, a user might choose `creator` for the Dailymotion video owner, and another user might choose `author` for the YouTube video owner. To ensure that users use the same types/properties for similar sites, a future direction would be to show previous descriptions of similar sites to the users and suggest they use the same schema.

Similarly, APIs that look similar still might not permit updating the same set of properties of a given object. One API might allow changing both the description and the name of a playlist and another might only allow changing the description. ShapirJS provides these details about which object's properties can be edited for each API. However, this forces an author to keep the source of a particular object in mind if they want to know which parts of it they can edit. A future improvement would be to add error handling when the user tries to edit a property that cannot be edited. In addition, similar web sites might provide different types. For example, YouTube, Vimeo and Dailymotion all provide `VideoObject` objects, but only YouTube and Vimeo provide `Comment` as an additional type that Dailymotion does not offer. If a user creates a Mavo application that shows YouTube and Dailymotion for videos and their comments, Shapir will peacefully handle this by showing comments with YouTube videos but none with Dailymotion videos since Dailymotion does not support the `Comment` type.

Another limitation the Shapir obscures but does not resolve is that each web site API is only designed for the data on that web site. For example ShapirJS allows a user to fetch videos and put them into playlists of a common schema.org type. But while the *schema* seems to allow placing a video from Dailymotion into a playlist from YouTube, ShapirJS would be unable to actually *execute* this operation because YouTube would not accept the Dailymotion video into its playlist. The user would therefore need to store such a"mixed origin" playlist locally if they created one. It is exciting to envision a future in which types have been standardized by WoOPI such that data from one web site can be stored in a different web site with the same schema.

11 CONCLUSION

This paper proposes WoOPI, a standardized, machine-readable ontology that can be used to simplify accessing web APIs. It presents Shapir, a system that includes the ShapirUI GUI to make WoOPI descriptions and the ShapirJS client library that presents all web site objects as objects in the application's local environment, which can be manipulated by getting and setting object properties or invoking apparently-local methods. We integrate ShapirJS with Mavo to allow non-programmers to access web APIs through their HTML-only Mavo applications. Our evaluations showed that programmers were able to access APIs and accomplish complex tasks using ShapirJS 5.6 times faster on average than using a well-known JavaScript library. Even non-programmers were able to create complete applications that access multiple web APIs in just 4 minutes using Shapir with Mavo.

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] Prompt API. 2021. Hassle-free API Marketplace. Retrieved July 18, 2021 from https://promptapi.com/
[3] Tim Berners-Lee, James Hendler, Ora Lassila, et al. 2001. The semantic web. *Scientific american* 284, 5 (2001), 28–37.
[4] API Blueprint. 2019. Retrieved August 10, 2019 from https://apiblueprint.org

[5] Dan Brickley, Matthew Burgess, and Natasha Noy. 2019. Google Dataset Search: Building a search engine for datasets in an open Web ecosystem. In *The World Wide Web Conference*. 1365–1375.

[6] Jill Cao, Kyle Rector, Thomas H Park, Scott D Fleming, Margaret Burnett, and Susan Wiedenbeck. 2010. A debugging perspective on end-user mashup programming. In *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 149–156.

[7] 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.

[8] 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.

[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] Giusy Di Lorenzo, Hakim Hacid, Hye-young Paik, and Boualem Benatallah. 2009. Data integration in mashups. *ACM Sigmod Record* 38, 1 (2009), 59–66.

[11] Milan Dojchinovski and Tomas Vitvar. 2018. Linked web APIs dataset. *Semantic Web* 9, 4 (2018), 381–391.

[12] 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.

[13] Robert J Ennals and Minos N Garofalakis. 2007. MashMaker: mashups for the masses. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 1116–1118.

[14] Brad Fitzpatrick and David Recordon. 2014. Thoughts on the social graph, 2007. *URL http://bradfitz. com/social-graph-problem/. Visited on* 18, 05 (2014).

[15] Wael H Gomaa, Aly A Fahmy, et al. 2013. A survey of text similarity approaches. *International Journal of Computer Applications* 68, 13 (2013), 13–18.

[16] GraphQL. 2019. GraphQL. Retrieved August 1, 2019 from https://graphql.org

[17] 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.

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

[19] 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.

[20] Austin Haugen. 2010. The open graph protocol design decisions. In *International Semantic Web Conference*. Springer, 338–338.

[21] Aidan Hogan. 2020. The semantic web: Two decades on. *Semantic Web* 11, 1 (2020), 169–185.

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

[23] David F Huynh, Robert C Miller, and David R Karger. 2007. Potluck: Data mash-up tool for casual users. In *The Semantic Web*. Springer, 239–252.

[24] IFTTT. 2021. IFTTT - Connect your apps and devices in new and remarkable ways. Retrieved July 18, 2021 from https://ifttt.com

[25] Walter N Kernan, Catherine M Viscoli, Robert W Makuch, Lawrence M Brass, and Ralph I Horwitz. 1999. Stratified randomization for clinical trials. *Journal of clinical epidemiology* 52, 1 (1999), 19–26.

[26] klazify. 2021. Klazify. Retrieved Marcth 27, 2021 from https://www.klazify.com

[27] 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.

[28] Node-RED. 2013. Retrieved August 15, 2019 from https://nodered.org

[29] Peter F Patel-Schneider. 2014. Analyzing schema. org. In *International Semantic Web Conference*. Springer, 261–276.

[30] Postman. 2019. Postman. Retrieved August 10, 2019 from https://www.getpostman.com

[31] ProgrammableWeb. 2005. ProgrammableWeb Search Category. Retrieved September 1, 2019 from https://www.programmableweb.com

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

[33] Amir Rahmati, Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2017. IFTTT vs. Zapier: A comparative study of trigger-action programming frameworks. *arXiv preprint arXiv:1709.02788* (2017).

[34] RAML. 2019. RAML. Retrieved August 10, 2019 from https://raml.org

[35] RapidAPI. 2021. RapidAPI. Retrieved March 20, 2021 from https://rapidapi.com

[36] Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009), 27–34.

[37] Swagger Specification. 2019. OpenAPI Specification. Retrieved August 15, 2019 from https://swagger.io/specification

[38] Swagger. 2019. Swagger. Retrieved August 10, 2019 from https://swagger.io

[39] Swagger. 2021. Swagger Client. Retrieved Marcth 29, 2021 from https://github.com/swagger-api/swagger-js

[40] Swagger. 2021. Swagger Editor. Retrieved Marcth 29, 2021 from https://swagger.io/tools/swagger-editor

[41] 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.

[42] 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.

[43] 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.

[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] Zapier. 2021. RapidAPI. Retrieved July 18, 2021 from https://zapier.com