Web Voyager: Lightweight Visualization Of Website Data

Kapaya Katongo
MIT CSAIL
Cambridge, MA, USA
kkatongo@mit.edu

ABSTRACT

Many websites have data that we would like to visualize but it is not available in a form that can be readily used such as a JSON or CSV file. The few websites that do make their data available provide it via an API which often requires setting up an account, registering for an API key and configuring a programming environment which can be time consuming and complex.

In this paper, I present an interaction model for lightweight visualization of structured website data right in the context of the website. The key idea is to provide a mechanism to scrape the desired data from the website and feed it into a visualization pipeline that enables lightweight visualization of the data without having to leave the website.

To illustrate this approach, I have implemented a Chrome browser extension called Web Voyager. Through case studies, I show that Web Voyager can be used to visualize data on real-world websites. Finally, I discuss the limitations of Web Voyager and opportunities for future work.

CCS CONCEPTS

• **Human-centered computing → Web-based interaction**; • **Software and its engineering → Integrated and visual development environments**.

KEYWORDS

data visualization, end-user web scraping

ACM Reference Format:
Kapaya Katongo. 2021. Web Voyager: Lightweight Visualization Of Website Data. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Many websites have data that we would like to visualize but it is often not available in a form that can be readily used such as a JSON or CSV file. Furthermore, the few websites that do make their data available provide it via an Application Programming Interface (API). This often requires setting up an account, registering for an API key and configuring a programming environment. This can not only be time consuming but also act as a barrier to people not familiar with the required programming expertise.

Web scraping has long been used to extract data from websites. It can be done via traditional programming by writing code [1–3], via programming-by-demonstration by providing examples of the data to scrape [6, 9, 10] and via automatic scraping by using heuristics to identify the data to scrape [7]. Web scraping via traditional programming is the most powerful but results in the same barrier as accessing data through an API: the user needs to have the required programming expertise. Automatic web scraping is the most convenient because it does not require any intervention from the user. However, the heuristics that make it possible are often domain specific and do not extend across a greater variety of websites. Web scraping via programming-by-demonstration provides that best of both: users provide examples of the data to scrape and the system automatically determines the entire set of data to scrape. In this paper, I present an interaction model for lightweight visualization of structured website data right in the context of the website. The key idea is to enable users to scrape the desired data from a website via programming-by-demonstration and feed it into a visualization pipeline that enables lightweight visualization of the data. This interaction model enables users to visualize website data right in the context of the website with the same ease of use as accessing the data via a JSON or CSV file.

To illustrate this approach, I implemented a Chrome browser extension called Web Voyager. It uses Vega-lite [11] to specify visualizations using the scraped data. Vega-lite was chosen over other visualization tools such as D3 [4] and Vega [12] because it provides a more concise and convenient form to author common visualizations. Section 2 shows a concrete scenario in which Web Voyager can be used to visualize data on a website that is only available via an API. In Section 3, I outline the implementation of the web scraping approach that makes website data available for visualization in the context of the website. To evaluate Web Voyager, I describe a suite of case studies on real world websites that it can be used on in Section 4. Furthermore, I explain the current limitations of the web scraping approach and the categories of websites that Web Voyager cannot be used on. In Section 5, I compare Web Voyager to existing tools that visualize website data that it relates to. Finally, Section 6 discusses opportunities for future work such as the ability to specify visualizations via direct manipulation without directly creating Vega-lite specifications.

2 EXAMPLE SCENARIO

Jen uses timeanddate.com to view a table of all the holidays and observances for a given year in the United States. She is curious about what day of the week the most federal holidays fall on and would like to visualize this using Vega-lite. timeanddate.com does not provide the holiday data as a JSON or CSV file but it does make

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

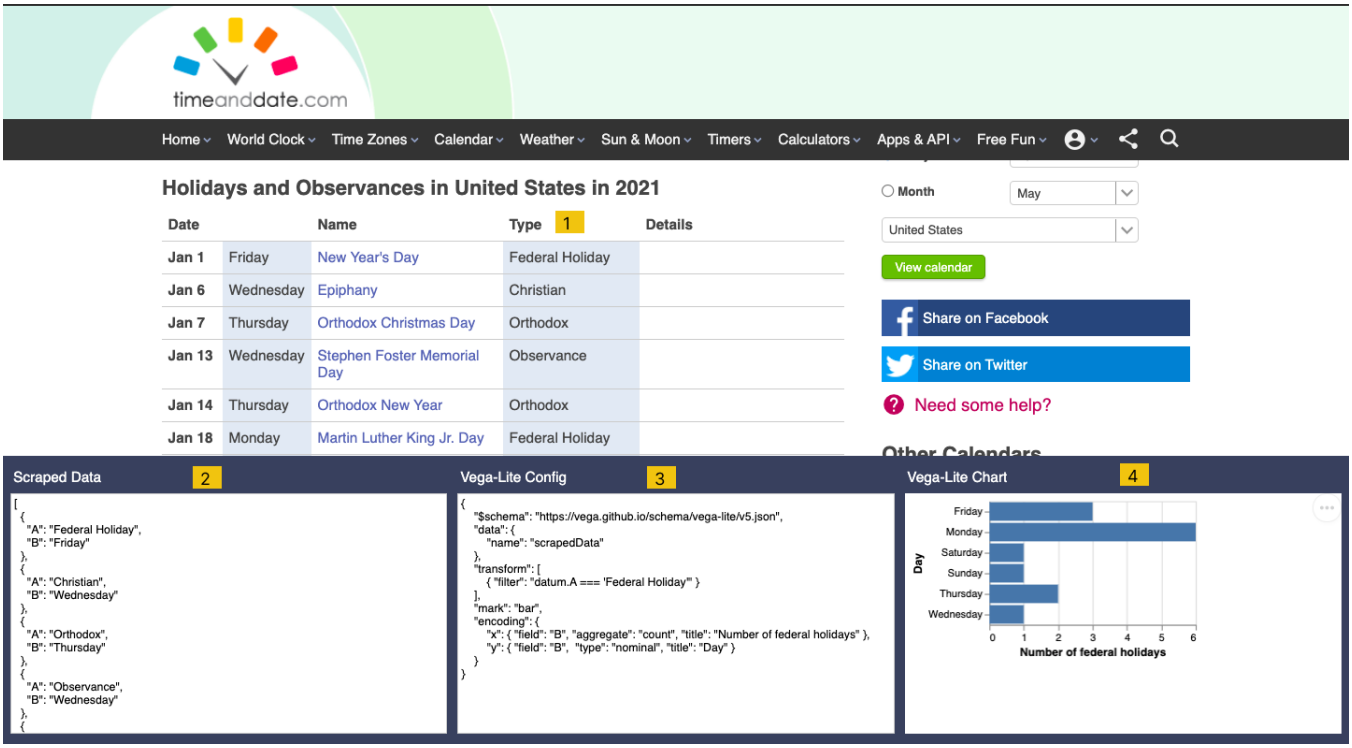

Figure 1: An overview of visualizing holiday data on timeanddate.com using Web Voyager. 1) Holiday type values scraped and highlighted (light blue) by clicking on a single holiday type value 2) Scraped holiday data displayed as a JSON array with auto generated property names 3)Vega-lite specification used to create the visualization 4) Visualization created from Vega-lite specification

it available via API. However, Jen does not have the time to register for an account, create an API key and set up a programming environment. Below, I show how Jen can use Web Voyager to visualize the holiday data with a few simple and easy steps right in the context of the website. The entire process is illustrated in Figure 1.

2.1 Scraping Data

Jen starts out by initiating Web Voyager through the browser context menu. This renders a panel with three sections: one for the data that will be scraped, one to specify the visualization and one to show the visualization. To visualize the number of federal holidays that occur on each day of the week, Jen will need to scrape the holiday type values and holiday day values. She clicks on the holiday type of the first row holiday row (New Years Data) and Web Voyager generalizes to the holiday types in all the other rows (Figure 1 Part 1). After the click, Web Voyager provides visual feedback about what values have been scraped by highlighting them as well as populating them into a JSON array in the *Scraped Data* section (Figure 1 Part 2) of the panel with an auto generated property name (*A*). Jen repeats the process to scrape the holiday day values.

2.2 Specifying & Rendering Visualization

Now that Jen has extracted the data she needs, she can build on the provided Vega-lite specification in the middle section to create a bar chart (Figure 1 Part 3). She starts by filtering out holiday rows whose federal holiday value (property A) is not equal to "Federal Holiday." Then, she specifies that the x axis should encode a count of the week day values (property B) and the y axis should encode the week day values as a nominal type. As she makes changes to the Vega-lite specification, Web Voyager automatically executes to render the visualization being specified if the specification is valid. Once complete, the specification leads to a bar chart that shows Monday has the most number of federal holidays (Figure 1 Part 4).

3 IMPLEMENTATION

This section focuses on the web scraping implementation as Vega-lite [11] is used as it would in any other programming environment.

When users demonstrate a value to scrape, Web Voyager must synthesize a program that reflects the user' general intent. This is an instance of the wrapper induction [8] problem of synthesizing a web data extraction query from examples. In the next two sections, I describe the two stages involved in this process.

3.1 Determining Row Elements

Given a DOM element v representing a value to scrape, we must find a set of a set of *row elements* that represent the rows of the containing data table. We could naively assume that $parent(v)$ is the

row containing v, but often v is deeply nested inside its containing row; we must determine which ancestor of v is likely to be the row.

Intuitively, we solve this problem by assuming that all rows share some similar internal structure. In particular, we expect most rows to contain a value for the demonstrated column. (If there were no missing data, we'd expect *all* rows to contain data for this column.)

Formally: assume a function $select(el, s)$ which runs a CSS selector that returns the set of elements matching s within el. We generate a set of plausible candidates P, consisting of pairs of a row element and a CSS selector:

$P = \{(r, s) \mid r \in ancestors(v) \wedge select(r, s) = \{v\}\}$

For each candidate $(r, s) \in P$, we compute a weight function w, which is based on the number of siblings of r that have "similar structure," defined by checking whether running s within the sibling also returns a unique element.

$w(r, s) = |\{r' \mid r' \in siblings(r) \wedge |select(r', s)| = 1\}|$

We then choose the candidate with the highest weight. In case of ties, the candidate closer to v in the tree (i.e., lower in the tree) wins. Given a winning candidate (r, s), the full set of row elements is $\{r\} \cup siblings(r)$.

3.2 Synthesizing CSS Selectors For Column Values

Once we have determined the row elements, next we must choose a CSS selector that will be used to identify the demonstrated value within its row.

Given a demonstrated value v within a row element r, we generate two kinds of plausible selectors:

- selectors using CSS classes, which are manual annotations on DOM elements added by the website's programmers, typically for styling purposes (e.g. "item__price")
- selectors using positional indexes within the tree, using the `nth-child` CSS selector (e.g. `nth-child(2)`, representing the second child of an element)

The minimum criteria for a plausible selector s is that it uniquely identifies the value within the row: $select(r, s) = \{v\}$. But there may be many plausible selectors, so we must pick a best one.

We first prioritize selectors using classes, because they tend to be more robust to changes on the website and are more readable. A single selector can combine multiple classes, but we prefer using fewer classes when possible. If no plausible class-based selector can be generated (for example, if the relevant elements don't have any classes to query), we fall back to using a positional index selector. This kind of selector can always be generated regardless of the contents of the page, but tends to be less accurate and robust.

4 EVALUATION

4.1 Case Studies

To evaluate Web Voyager, I used it to visualize data on real world websites. For the websites on which it can be used, I provide the sequence of steps needed to create the visualization. For the websites on which Web Voyager fails, I explain the relevant limitations.

4.1.1 Google Scholar. On the Google Scholar page of a given author, Web Voyager can be used to visualize how many publications per year another author appeared as a co-author. This requires scraping the publication author and publication year values which can be done with just two clicks. Then, a new property can be created (via the calculate transform) which indicates whether an author appears in the list of a publication's authors. Finally, a chart can be specified that encodes a count of the created property on the x-axis and the publication year on the y-axis.

4.2 Limitations

Web Voyager' wrapper induction process is most effective on websites whose data is presented as a collection of similarly-structured HTML elements. Certain websites, however, have designs that make it difficult to scrape data:

Heterogeneous row elements. Some websites break their content into rows, but the rows do not have a consistent layout, and contain different types of child elements. For example, the page design of HackerNews alternates between rows containing a title and rows containing supplementary data (e.g. number of likes and the time of posting). Because MWS only chooses a single row selector, when extracting by demonstration, Joker will only select one of the types of rows, and elements in the other types of rows will not be extracted.

Infinite scroll. Some websites have an "infinite scroll" feature that adds new entries to the page when a user scrolls to the bottom. Joker's table will only contain elements that were rendered when the table was first created.

Large number of DOM elements. For websites that render a very large number of DOM elements, the speed of the data highlighting significantly decreases. This is because the wrapper induction process queries the DOM which takes longer as the size of the DOM increases.

5 RELATED WORK

Web Voyager builds on existing work in end-user scraping and data visualization across a number of tools and systems.

5.1 End-user Web Scraping

The wrapper induction approach used by Web Voyager is inspired by that of Vegemite [10], a tool for end-user programming of web mashups. Like Web Voyager, Vegemite enables end-users to scrape data from a website into a tabular format via a simple mouse click. Once scraped, the data can be used as part of a web automation task such as sending an email to a column of scraped email addresses. Similar web scraping approaches can be seen in tools like Rousillon [6] and FlashExtract [9].

5.2 Data Visualization

Web Voyager uses Vega-lite [11] to enable users to visualize the scraped data. Vega-lite is a high level grammar of interactive graphics which provides a concise, declarative JSON syntax for specification visualizations. Once a website's data is scraped into a JSON array, it is made available to a Vega-lite specification, via the data property, which can be used to create a visualization.

More broadly, Web Voyager relates to other tools [5, 13] that enable visualization of website data, of which DS.js [15] is the

most similar. DS.js is a bookmarklet that embeds a data science programming environment into websites. It does this by automatically parsing HTML tables and CSV links and then embedding code editors with live previews of the derived data and visualizations. While DS.js provides a more extensive set of features for visualizing website data, it can only scrape data in HTML tables and CSV links. Web Voyager can scrape any data on a website that appears in a structured format and the ability to scrape data from CSV links can easily be added.

6 CONCLUSION AND FUTURE WORK

In this paper, I present an interaction model for lightweight visualization of structured website data right in the context of the website. This interaction model enables users to visualize website data right in the context of the website with the same ease of use as accessing the data via a JSON or CSV file.

The main area of future work is further reducing the barrier between users and a visualization of website data. The current approach requires users to be familiar with Vega-lite. While Vega-lite makes it possible to create a range of useful visualizations in a concise and simple manner, the kind of lightweight visualizations users have in mind might not warrant having to learn the syntax. Voyager 2 [14] provides some inspiration for automatically creating Vega-lite specifications that I will explore in the next iteration of Web Voyager.

My ultimate goal is to enable anyone that uses the web to visualize website data in the course of their daily use in an intuitive way.

REFERENCES

[1] [n.d.]. *Beautiful Soup: We Called Him Tortoise Because He Taught Us.* https://www.crummy.com/software/BeautifulSoup/

[2] [n.d.]. *Scrapy | A Fast and Powerful Scraping and Web Crawling Framework.* https://scrapy.org/

[3] [n.d.]. *SeleniumHQ Browser Automation.* https://www.selenium.dev/

[4] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. [n.d.]. D³ Data-Driven Documents. 17, 12 ([n. d.]), 2301–2309. https://doi.org/10.1109/TVCG.2011.185

[5] Bryan Chan, Leslie Wu, Justin Talbot, Mike Cammarano, and Pat Hanrahan. [n.d.]. Vispedia: Interactive Visual Exploration of Wikipedia Data via Search-Based Integration. 14, 6 ([n. d.]), 1213–1220. https://doi.org/10.1109/TVCG.2008.178

[6] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. [n.d.]. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin Germany, 2018-10-11). ACM, 963–975. https://doi.org/10.1145/3242587.3242661

[7] David F. Huynh, Robert C. Miller, and David R. Karger. [n.d.]. Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology - UIST '06* (Montreux, Switzerland, 2006). ACM Press, 125. https://doi.org/10.1145/1166253.1166274

[8] Nicholas Kushmerick. [n.d.]. Wrapper Induction: Efficiency and Expressiveness. 118, 1-2 ([n. d.]), 15–68. https://doi.org/10.1016/S0004-3702(99)00100-9

[9] Vu Le and Sumit Gulwani. [n.d.]. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh United Kingdom, 2014-06-09). ACM, 542–553. https://doi.org/10.1145/2594291.2594333

[10] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. [n.d.]. End-User Programming of Mashups with Vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces* (Sanibel Island Florida USA, 2009-02-08). ACM, 97–106. https://doi.org/10.1145/1502650.1502667

[11] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. [n.d.]. Vega-Lite: A Grammar of Interactive Graphics. 23, 1 ([n. d.]), 341–350. https://doi.org/10.1109/TVCG.2016.2599030

[12] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. [n.d.]. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. 22, 1 ([n. d.]), 659–668. https://doi.org/10.1109/TVCG.2015.2467091

[13] Michael Toomim, Steven M. Drucker, Mira Dontcheva, Ali Rahimi, Blake Thomson, and James A. Landay. [n.d.]. Attaching UI Enhancements to Websites with End Users. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems - CHI 09* (Boston, MA, USA, 2009). ACM Press, 1859. https://doi.org/10.1145/1518701.1518987

[14] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. [n.d.]. Voyager 2: Augmenting Visual Analysis with Partial View Specifications. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver Colorado USA, 2017-05-02). ACM, 2648–2659. https://doi.org/10.1145/3025453.3025768

[15] Xiong Zhang and Philip J. Guo. [n.d.]. DS.Js: Turn Any Webpage into an Example-Centric Live Programming Environment for Learning Data Science. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City QC Canada, 2017-10-20). ACM, 691–702. https://doi.org/10.1145/3126594.3126663