





A Data Service Layer for Web Browser Extensions

Alex Tacuri¹^a, Sergio Firmenich^{1,2}^b, Gustavo Rossi^{1,2}^c and Alejandro Fernandez¹^d

¹LIFIA, CIC, Facultad de Informática, UNLP, Argentina

²CONICET, Argentina

fi

Keywords: Web Scraping, Web Browser Extensions.

Abstract: Web browser extensions are the preferred method for end-users to modify existing web applications (and the browser itself) to fulfill unanticipated requirements. Some extensions improve existing websites based on online data, combining techniques such as mashups and augmentation. To obtain data when no APIs are available, extension developers resort to scraping. Scraping is frequently implemented with hard-coded DOM references, making code fragile. Scraping becomes more difficult when a scraping pipeline involves several websites (i.e., the result of scraping composes elements from various websites). It is challenging (if not impossible) to reuse the scraping code in different browser extensions. We propose a data service layer for browser extensions. It encapsulates site-specific search and scraping logic and exposes object-oriented search APIs. The data service layer includes a visual programming environment for the specification of data search and object model creation, which are exposed then as a programmatic API. While using this data service layer, developers are unconcerned with the complexity of data search, retrieval, scraping, and composition.

1 INTRODUCTION

The web browser has evolved beyond being a mere client for displaying web pages. It has transformed into a robust platform capable of running feature-rich applications known as web browser extensions (referred to interchangeably as browser extensions or simply extensions). These extensions serve to enhance websites and augment the web browser with additional functionalities. For instance, extensions can introduce new capabilities to the browser, such as web scraping. They can also enhance accessibility, integrate crypto-wallets, modify the appearance or behavior of loaded web pages, customize the default behavior of the new tab page, or even serve as a platform for hosting specific applications within the web browser.

Numerous browser extensions rely on online content, often utilizing web scraping techniques to retrieve the necessary information. Alternatively, some extensions leverage APIs to access specific data elements. For example, there are multiple browser extensions designed to provide alerts regarding cryptocur-

rency fluctuations. These extensions typically scrape information from online crypto portals or utilize APIs to access real-time data.


In various other instances, information is scraped from web pages and integrated into web augmentation artifacts or mashup applications (Díaz and Arellano, 2015).


While the availability of APIs has increased over the years, web scraping remains a widely employed technique for extracting web content, primarily due to the lack of APIs provided by many existing websites. However, web scraping poses several challenges.


Firstly, scraping code often relies on hardcoded DOM references, making it susceptible to breakages when websites undergo modifications. Any changes to the website's structure can render the scraping code ineffective.


Secondly, scraping becomes more complex when the process involves multiple websites. For instance, scraping pipelines that aggregate data from various sources requires additional effort to handle and integrate the scraped elements effectively.

Thirdly, reusability of scraping code across different browser extensions is often challenging, if not impossible. The specific requirements and contexts of each extension may differ, necessitating modifications or customizations to the scraping code for

^a <https://orcid.org/0000-0003-3159-5556>

^b <https://orcid.org/0000-0001-9502-2189>

^c <https://orcid.org/0000-0002-3348-2144>

^d <https://orcid.org/0000-0002-7968-6871>

each use case. Overall, while web scraping remains a prevalent technique for content retrieval, its implementation and maintenance can present difficulties related to website changes, multi-site scraping, and code reusability in different browser extensions.

This paper introduces a data service layer designed specifically for browser extensions. This layer serves as an encapsulation of site-specific search and scraping logic, offering object-oriented search APIs. A key component of the data service layer is a visual programming environment, which enables users to specify data search and object model creation. These specifications are then exposed as programmatic APIs.

Users have the ability to define a search API that leverages the search functionality of an existing website and specifies how search results should be abstracted into objects. Furthermore, users can combine multiple search APIs to construct more complex object models. When a search request is made to a composed API, it automatically triggers requests to its constituent APIs and establishes connections between the resulting objects, ultimately returning a graph of interconnected objects.

By utilizing these APIs, extension developers can create browser extensions without being burdened by the complexities of data search, retrieval, scraping, and composition. The data service layer streamlines the process and enables developers to focus on building extensions based on the APIs created through this layer.

The data service layer serves as a crucial intermediary between a website's DOM and the implementation of extensions. By separating the search APIs and object models from the website's structure, the layer significantly simplifies the maintenance process when websites undergo changes. With this approach, browser extensions can continue functioning without requiring immediate modifications.

The potential impact of this proposed data service layer is substantial, particularly considering the vast number of existing browser extensions and user scripts available in public repositories. Just focusing on Google Chrome Browser, there are currently over 137,345 extensions available. Furthermore, web augmentation artifacts such as user scripts have their own extensive repositories, with platforms like greasyfork.org hosting thousands of artifacts. Integrating these existing artifacts with the data service layer could greatly enhance their functionality and ease the development of new extensions.

This paper is structured as follows. Section 2 presents a motivating example. Section 3 presents the approach, its architecture, the tool involved in

creating search APIs, and the tool used for defining information models. Section 3.4 briefly presents a usability study. Section 5 introduces related works, and finally, Section 6 presents conclusions and future works.

2 MOTIVATING EXAMPLE

Figure 1 depicts a UI mockup of a browser extension that provides a mashup application for the domain of scientific research. The development of the mentioned mashup application involves integrating multiple components and implementing web scraping techniques. Here is a breakdown of the steps involved:

- Utilize the search engine from Springer: The browser extension needs to incorporate the search functionality provided by Springer. This involves making requests to the Springer search engine and retrieving the results.
- Parse the DOM of the results page: Once the search results are obtained from Springer, the browser extension needs to parse the HTML DOM (Document Object Model) of the results page to extract relevant information about the articles, such as their titles and other details.

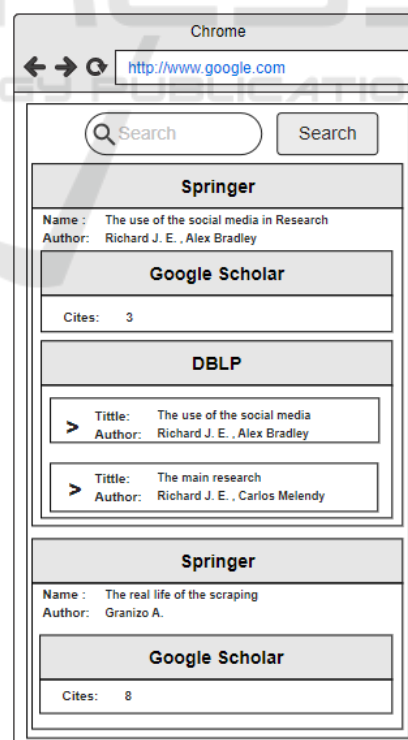


Figure 1: Mockup of a browser extension to mashup results from Springer, Google Scholar, and DBLP.

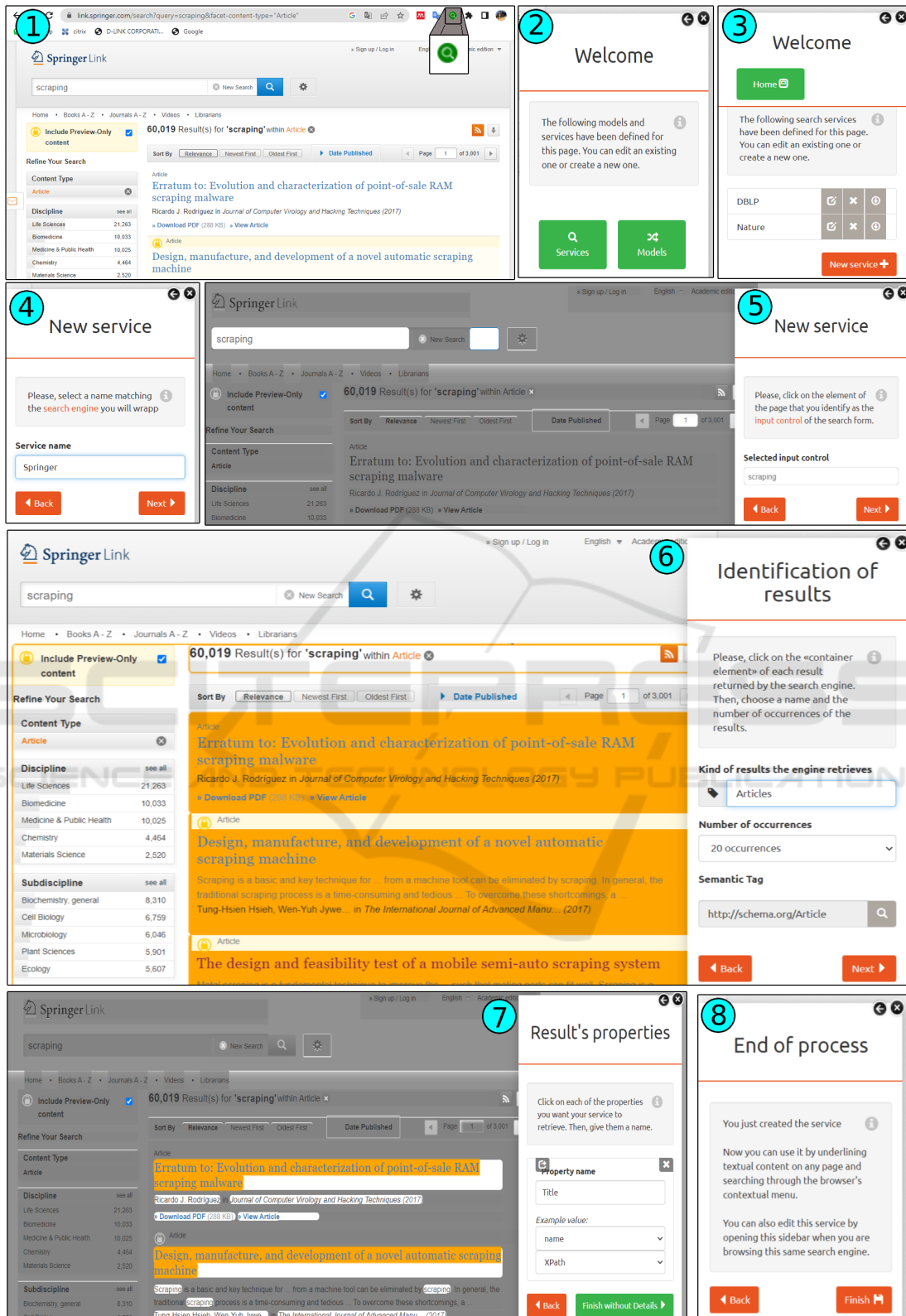


Figure 2: Procedure to create the Springer search API.

- Retrieve the number of citations from Google Scholar: After selecting an article from Springer, the extension needs to extract the article’s title and use it to search for the same article on Google Scholar. By parsing the DOM of the Google Scholar results page, the number of citations for the article can be retrieved.
- Obtain related articles from DBLP: Similarly, when an article’s author is selected, the extension needs to extract the author’s information and use it to search for related articles on DBLP. By scraping the relevant data from the DBLP website, the extension can retrieve and display the related articles.

To accomplish these tasks, the browser extension developer must create custom scrapers for all three websites: Springer, Google Scholar, and DBLP. Additionally, the developer must program the logic to establish the pipeline of data retrieval and processing. This includes passing the article title from Springer to Google Scholar and the author’s information from Springer to DBLP.

3 THE APPROACH AND TOOLS

In this section, we present the approach and tools. First, we provided an overview of the ANDES approach to create search APIs based on web page annotations. Then, we describe the architecture of our proposal. Then, the model editor is presented, which makes it possible to compose different search APIs to create more complex information models.

3.1 ANDES Search APIs

This work is based on ANDES’ concept of search APIs as reported in (removed for double blind revision, 2022). By means of web annotation and scraping techniques, ANDES abstracts the search functionality of any website into a reusable search API. Calling the search operation on the API triggers the corresponding search and scraping behavior and returns a collection of (JavaScript) objects. ANDES offers tools to annotate the UI search components (search input, trigger button, pagination buttons, etc.) and to abstract the search results as domain objects. The user defines the type of the resulting object, its properties, and the mapping from the content of the results web page to property values.

Further details about ANDES can be found in (removed for double blind revision, 2022). For the sake of comprehension, Figure 2 depicts the main parts

of the procedure to create a search API for Springer (as required for the motivating example). The user conducts a search on the website of interest (in this case, Springer’s website) and, once results are displayed, clicks on the browser extension icon (Step 1). The main UI of the ANDES’ browser extension opens. She selects the ”Services” button (Step 2) and clicks on the ”New service” button (Step 3). She assigns a name to the new search service, in this case, ”Springer” (Step 4). Then, the user selects the text input area in the search form using point-and-click interaction (Step 5). Once the text input has been selected, she chooses a UI element (a DOM node) containing one result. Finally, she defines a semantic type for the result and the repetition strategy that correctly collects all results available on the sample page (Step 6). Behind the scenes, each option in this menu represents an xPath used to retrieve all the results. Once result elements are identified, the user connects parts (sub-nodes in the DOM) of the result element to properties (Step 7).

3.2 Architecture

The data service layer and related editors are packaged as a standalone browser extension. Once installed in a web browser, the editors can be used to define both the search APIs for specific websites and the composition models to create more complex information objects. Both kinds of artifacts will become available for other browser extensions.

Figure 3 shows an overview of the architecture. It shows that the data service layer includes two tools (Search API tool and API composition tool), each one generating a kind of artifact that is stored in this layer. It also provides an API Interaction service layer that

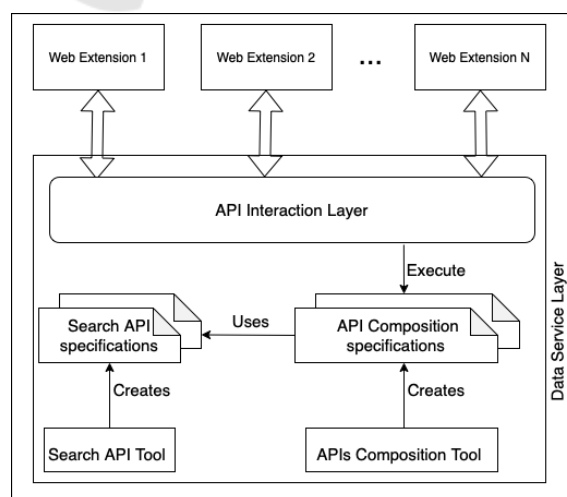


Figure 3: Data service layer architecture.

allows other browser extensions to consume the information models created with the API composition tool.

Figure 4 presents a high-level design of the data service layer. Search APIs are exposed through instances of SearchModel, managed by the ModelRegistry. Instances of SearchModel implement the search() operation called by the clients of the data service layer. In the most interesting case¹, a SearchModel has ComposedSearchAPI that drives the main search and composition task. In turn, the ComposedSearchAPI has one BasicSearchAPI (its main search) and one or more Augmenters. Given an object (a result from the main search), an Augmenter builds a search query, conducts a search on another search API, and injects the results as a new object property. The collaboration will become clearer in the following sections.

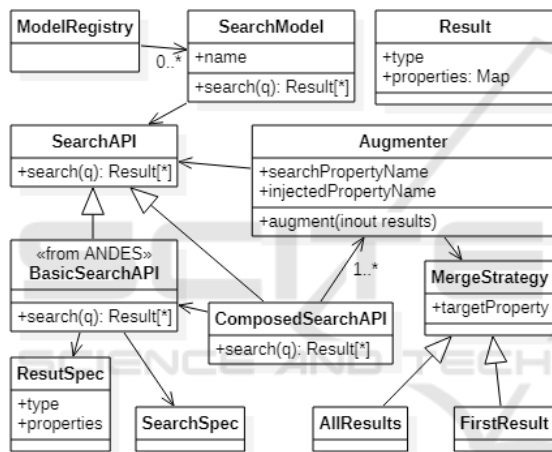


Figure 4: High-level design of the data service layer.

3.3 Search API Composition

To better understand how the data service layer is used to create browser extensions, let us return to the motivating example presented in Section 2. The main idea is to obtain a search model based on the composition of search APIs for Springer, DBLP, and Google Scholar. The model should integrate information from the three websites in a single response. Specifically, when the model is asked to search for something, it uses the Springer search service. In addition, the response also includes the number of citations obtained from Google Scholar and a collection

¹In the simplest case, the SearchModels has only a BasicSearchAPI, which results in a functionality comparable to what was already provided by ANDES as described in Section 3.1

of related articles (articles from the first author) obtained from DBLP.

To create an information model based on search service composition, the user creates the search services of the websites that he wishes to integrate, following the process presented in Section 3.1. Then, the user starts the API Composition Tool from the extension’s main UI. Next, the extension shows an editor (Figure 7) where the defined search services are presented in a panel on the left, and the composition canvas is the main component of the UI. The user drags&drops the services he wants to compose (i.e., Springer, Google Scholar, and DBLP services) into the canvas. Each search service is shown as a node of a graph. To integrate the results of the search services, the user creates links between the nodes. A link indicates that each result in the origin search service (e.g., Springer) will have a property holding related results in the target search service (e.g., DBLP). The link is configured with a) the name of the property holding the results (e.g., “related-articles”), b) a property in the model of the origin service whose value will be used as a query string to search in the destination service (e.g., “first-author”), and c) the strategy used to select which results to keep (e.g., “all results”). In this way, the integration between services can be one-to-one (connect to the object corresponding to the first or to any search result) or one-to-N (connect to a collection with all the results). Figure 5 depicts the resulting object model, following the design presented in Figure 4. The SearchModel relies on a ComposedSearchAPI that starts with a BasicSearchAPI (on Springer) and uses two Augmenters. One of them injects results obtained from a BasicSearchAPI configured on DBLP, and the other one injects results from a BasicSearchAPI configured on Google Scholar. Figure 6 provides additional details regarding the interactions occurring when the search() operation is called on a SearchModel object.

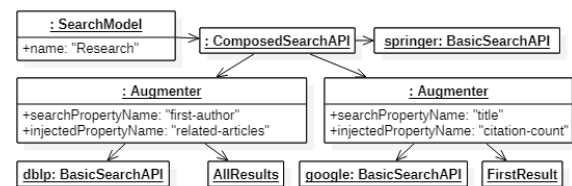


Figure 5: Object diagram corresponding to the motivating example.

3.4 Using the Data Service Layer

In this section, two use cases are presented. The first one corresponds to the mashup application presented as the motivating example in Section 2. The sec-

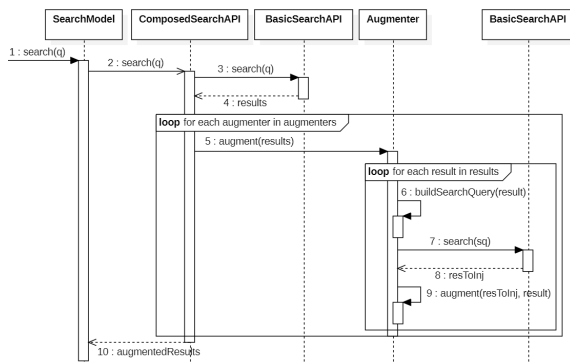


Figure 6: Interaction diagram corresponding to the motivating example.

and one is a web augmenter that augments Springer’s website with more information about a specific paper. Both examples use the same search model, which was presented in the previous section.

3.4.1 A Mashup Application

Consider the motivating example discussed in Section 2. The browser extension presents a main screen (see the browser window in the middle of Figure 8) with an input text, where the user enters a topic to search for (in this example, it was “scraping”). The search button triggers a search using the search model. Results are displayed along with the number of citations (which is extracted from Google Scholar) and related articles of the selected author (extracted from DBLP). Next, we discuss key parts of the source code (see Listing 1) to show that the developer remains unconcerned about the underlying web requests and scraping tasks.

To trigger the search, the mashup developer uses the data service layer whose API has the “search” method, shown in line 2. This method receives several parameters: the text to search, the name of the target search model. When the service layer is invoked through the search function, it executes the model and returns a JSON object, whose structure is composed of the information extracted from the three websites.

```

1 $scope.searchData = function ( args )
2   {
3     var api=new dataServiceLayer();
4     api.search( args.searchQuery ,
5               args.searchModelName ). then(
6               papers=>{
7                 $scope.results=papers
8               })
9   }
  
```

Listing 1: Source code for the service layer invocation.

3.4.2 Springer Website Augmenter

Consider the augmentation of Springer’s website depicted in Figure 9. The website is augmented to show further information about the paper that the user is navigating. In this case, the augmenter adds the number of citations (extracted from Google Scholar) and also further articles in which the first author has participated (extracted from DBLP).

Note that the code the developer must write to conduct searches and obtain the results is the same as in the mashup application previously discussed. The data service layer not only hides the complexity of scraping data from several websites and integrating this data but also supports reuse among different browser extensions.

4 PRELIMINARY EVALUATION

We conducted a usability study involving 16 participants to assess the tool’s usability and identify any usability issues. The intention is to subsequently conduct a more comprehensive experiment to examine the impacts and applicability of the approach thoroughly. The method for creating basic search APIs described in Section 3.1 was evaluated in (removed for double blind revision, 2022).

In this new study, participants were asked to do 7 tasks related to the creation and edition of API composition models. The model that participants were asked to create corresponds to the motivating example, using Springer, Google Scholar, and DBLP. 13 participants managed to complete the tasks, 2 participants completed 4 tasks, and 1 participant completed 5 tasks. It is important to mention that participants had no programming skills.

Upon completing the tasks, we utilized the System Usability Scale (SUS) method to assess usability, which yielded an average score of 68.11. Based on various standards, this score is considered acceptable.

5 RELATED WORKS

Web scraping means extracting non-structured (or partially structured) data from websites, often simulating the browsing behavior. Normally, it is used to automate data extraction to obtain more complex information, which means that end users are not usually involved in determining what information to look for and still less about what to do with the abstracted objects.

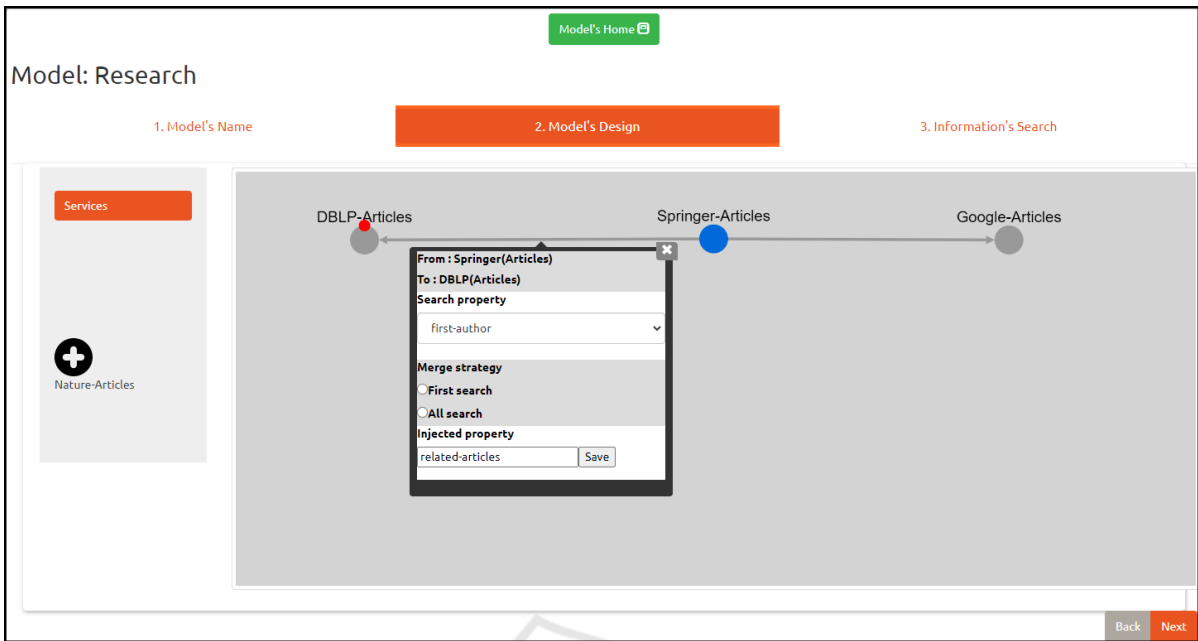


Figure 7: The API composition tool: Integration between Springer and DBLP search APIs.

Figure 8: Search results in the model containing Springer, Google Scholar and DBLP.

Some websites already tag their contents allowing other software artifacts (for instance, a web browser

plugin) to process those annotations and improve interaction with that structured content. A well-known

The screenshot shows the SpringerLink website interface. At the top, there is a search bar and a 'Log in' button. The breadcrumb trail indicates the article is in the 'Cluster Computing' section. The main title is 'Correction: Static to dynamic transition of RPL protocol from IoT to IoV in static and mobile environments', published on 23 August 2022, by Sakshi Garg, Deepti Mehrotra, Hari Mohan Pandey, and Sujata Pandey. The article is from 'Cluster Computing', volume 26, issue 863 (2023). A 'Download PDF' button is visible. A 'Working on a manuscript?' section offers to avoid common mistakes. A 'Sections' list includes 'Correction: Cluster Computing https://doi.org/10.1...', 'Author information', 'Additional information', 'Rights and permissions', and 'About this article'. A 'Ver PDF' button is also present. A 'nature' advertisement is shown at the bottom.

Figure 9: Reusing the search API to augment Springer's website.

approach for giving some meaning to web data is Microformats (Khare and Çelik, 2006). Some approaches leverage the underlying meaning given by Microformats, detecting those objects present on the web page and allowing users to interact with them in new ways. A very similar approach is Microdata. Considering Semantic web approaches and an aim similar to our proposal, (Kalou et al., 2013) presents an approach for mashups based on semantic information; however, it depends too heavily on the original application owners, something that is not always viable. Moreover, when analyzing the Web, we see that a the majority of websites do not provide this annotation layer. According to (Bizer et al., 2013), only 5,64% among 40.6 million websites provide some kind of structured data (Microformats, Microdata, RDFa, etc.). This reality emphasizes the significance of empowering users to incorporate semantic structure in situations where it is lacking.

Several approaches let users specify the structure of existing contents to ease the management of relevant information objects. For instance, HayStack (Karger et al., 2005) offers an extraction tool that allows users to populate a semantic-structured information space. Atomate it! (Kleek et al., 2010) offers a reactive platform that could be set to the collected objects by means of rule definitions. Then the user can be informed when something interesting happens (e.g., a new movie is available). (Van Kleek et al., 2012) allows the creation of domain-specific applications that work over the objects defined in a PIM. Rousillon is an interesting approach based on end-user programming for defining scrapers based on hierarchical data (Chasins et al., 2018). In (Katongo

et al., 2021), another approach is presented to enable web customization through web scraping defined by users without programming skills. However, it differs from our approach because many browser extensions can use our data service layer.

Web augmentation is a popular approach that lets end users improve web applications by altering original web pages with new content or functionality not originally contemplated by their developers. Nowadays, users may specify their own augmentations by using end-user programming tools.

In previous work, the ANDES approach (removed for double blind revision, 2022) was introduced as a means of enabling end-user programming of search APIs based on the search functionality of websites. While the current paper builds upon ANDES, two noteworthy contributions distinguish it from the previous work. Firstly, ANDES primarily focuses on ancillary searches, which involve using the created search APIs to augment websites with the results of specific ancillary searches. In contrast, the present work extends the concept of search APIs by integrating them into a data service layer. This means that the search APIs are now offered as part of a broader framework that provides various data services. In comparison with ANDES, the second significant contribution of this paper is the proposed method and tool for composing search APIs. They allow for the creation complex information objects whose internal structures are populated with data retrieved and scraped from multiple sources. By enabling the composition of search APIs, developers can leverage information from diverse sources to construct comprehensive information models.

Other interesting tools have emerged in the context of end-user programming and web augmentation (Díaz et al., 2014). Reuse in web augmentation has also been tackled before. For example, Scripting Interface (Díaz et al., 2010) is oriented to better support reuse by generating a conceptual layer over the DOM, specifically for user scripts. Since the specification of a scripting interface could be defined in two distinct websites, the augmentation artifacts written in terms of that interfaces could be reused. However, these scripting interfaces do not consider the search engines provided by web applications.

6 CONCLUSIONS AND FUTURE WORKS

In this article, we highlight the unique nature of browser extensions as software tools that enable users to customize their web browsing experience. While some browser extensions can be developed without programming skills using end-user development tools, the complexity increases when scraping is involved.

The article introduces a data service layer approach to simplify the development process for browser extensions that require web scraping. The approach enables developers to define web scraping operations in a no-code manner by automating the interaction with search engines, effectively creating search services or APIs for websites that do not offer them. These search APIs can then be combined to create more complex information models, allowing developers to retrieve data from multiple websites in a single invocation.

An exciting outcome of the presented approach and tool is its potential integration into other browser extensions that aim to provide end-user programming capabilities. Incorporating this data service, including the end-user programming tools for defining search services and APIs composition, can serve as a foundational layer within a broader end-user programming tool for more general-purpose artifacts such as mashups or web augmenters. In addition, this integration would empower users to leverage web scraping capabilities seamlessly, enhancing their ability to customize and extend the functionality of browser extensions.

By embedding the data service layer into other extensions, developers can provide a comprehensive environment that combines the benefits of end-user programming with the flexibility of web scraping. This would enable a wider range of users to create sophisticated browser extensions that rely on scraping and

aggregating information from various websites.

A usability study was conducted to assess the effectiveness of the approach. While some usability issues were identified, the majority of participants, even those without programming skills, were able to define the necessary API compositions for the given tasks, demonstrating promising results.

As part of future work, the authors plan to conduct a more comprehensive experiment to validate other aspects of the data service layer approach. This will contribute to further refining and enhancing the usability and effectiveness of the proposed method.

REFERENCES

- Bizer, C., Eckert, K., Meusel, R., Mühleisen, H., Schumacher, M., and Völker, J. (2013). Deployment of rdfa, microdata, and microformats on the web - A quantitative analysis. In Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J. X., Aroyo, L., Noy, N. F., Welty, C., and Janowicz, K., editors, *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*, volume 8219 of *Lecture Notes in Computer Science*, pages 17–32. Springer.
- Chasins, S. E., Mueller, M., and Bodík, R. (2018). Rousillon: Scraping distributed hierarchical web data. In Baudisch, P., Schmidt, A., and Wilson, A., editors, *The 31st Annual ACM Symposium on User Interface Software and Technology, UIST 2018, Berlin, Germany, October 14-17, 2018*, pages 963–975. ACM.
- Díaz, O. and Arellano, C. (2015). The augmented web: Rationales, opportunities, and challenges on browser-side transcoding. *ACM Trans. Web*, 9(2):8:1–8:30.
- Díaz, O., Arellano, C., Aldalur, I., Medina, H., and Firmenich, S. (2014). End-user browser-side modification of web pages. In Benatallah, B., Bestavros, A., Manolopoulos, Y., Vakali, A., and Zhang, Y., editors, *Web Information Systems Engineering - WISE 2014 - 15th International Conference, Thessaloniki, Greece, October 12-14, 2014, Proceedings, Part I*, volume 8786 of *Lecture Notes in Computer Science*, pages 293–307. Springer.
- Díaz, O., Arellano, C., and Iturrioz, J. (2010). Interfaces for scripting: Making greasemonkey scripts resilient to website upgrades. In Benatallah, B., Casati, F., Kappel, G., and Rossi, G., editors, *Web Engineering, 10th International Conference, ICWE 2010, Vienna, Austria, July 5-9, 2010. Proceedings*, volume 6189 of *Lecture Notes in Computer Science*, pages 233–247. Springer.
- Kalou, A. K., Koutsomitropoulos, D. A., and Papatheodorou, T. S. (2013). Semantic web rules and ontologies for developing personalised mashups. *Int. J. Knowl. Web Intell.*, 4(2/3):142–165.
- Karger, D. R., Bakshi, K., Huynh, D., Quan, D., and Sinha, V. (2005). Haystack: A general-purpose information

- management tool for end users based on semistructured data. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 13–26. www.cidrdb.org.
- Katongo, K., Litt, G., and Jackson, D. (2021). Towards end-user web scraping for customization. In Church, L., Chiba, S., and Boix, E. G., editors, *Programming '21: 5th International Conference on the Art, Science, and Engineering of Programming, Cambridge, United Kingdom, March 22-26, 2021*, pages 49–59. ACM.
- Khare, R. and Çelik, T. (2006). Microformats: a pragmatic path to the semantic web. In Carr, L., Roure, D. D., Iyengar, A., Goble, C. A., and Dahlin, M., editors, *Proceedings of the 15th international conference on World Wide Web, WWW 2006, Edinburgh, Scotland, UK, May 23-26, 2006*, pages 865–866. ACM.
- Kleek, M. V., Moore, B., Karger, D. R., André, P., and m. c. schraefel (2010). Atomate it! end-user context-sensitive automation using heterogeneous information sources on the web. In Rappa, M., Jones, P., Freire, J., and Chakrabarti, S., editors, *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 951–960. ACM.
- removed for double blind revision, R. (2022). Title removed for double blind revision. *Comput. Stand. Interfaces*.
- Van Kleek, M., Smith, D. A., Shadbolt, N., et al. (2012). A decentralized architecture for consolidating personal information ecosystems: The webbox. In *PIM 2012*.

