Automated Information Extraction from Web APIs Documentation

Papa Alioune Ly[1,2], Carlos Pedrinaci[1], John Domingue[1]

[1] Knowledge Media Institute, The Open University
[2] School of Computer and Communication Sciences,
École Polytechnique Fédérale de Lausanne (EPFL)
{alioune.ly, carlos.pedrinaci, john.domingue}@open.ac.uk

Abstract. A fundamental characteristic of Web APIs is the fact that, de facto, providers hardly follow any standard practices while implementing, publishing, and documenting their APIs. As a consequence, the discovery and use of these services by third parties is significantly hampered. In order to achieve further automation while exploiting Web APIs we present an approach for automatically extracting relevant technical information from the Web pages documenting them. In particular we have devised two algorithms that automatically extract technical details such as operation names, operation descriptions or URI templates from the documentation of Web APIs adopting either RPC or RESTful interfaces. The algorithms devised, which exploit advanced DOM processing as well as state of the art Information Extraction and Natural Language Processing techniques, have been evaluated against a detailed dataset exhibiting a high precision and recall–around 90% for both REST and RPC APIs–outperforming state of the art information extraction algorithms.

Keywords: Web API, RESTful service, Web Page Segmentation, Information Extraction, Service Discovery

1 Introduction

On the Web, service technologies are currently marked by the proliferation of Web APIs, also called RESTful services when they conform to REST principles [1]. Major Web sites such as Facebook, Flickr or Amazon provide access to their data and functionality through Web APIs. This trend is impelled by the simplicity of the technology stack, compared to WS-* Web services [2], as well by the simplicity with which such APIs can be offered over preexisting Web sites infrastructure [3]. Thanks to these APIs, we have seen in the last years a proliferation of applications and Web sites that exploit and combine them to provide added-value solutions to users.

When building a new service-oriented application, it is fundamental to be able to swiftly discover existing services or APIs, to figure out what operations and resources they offer as well as to understand the ways in which one can use them. Unfortunately, supporting the aforementioned steps over the Web APIs one can

find on the Web is most challenging nowadays since most often the only way to carry this out requires interpreting highly heterogeneous HTML pages that are intended for humans and that provide no convenient means for supporting their identification and interpretation by machines. In fact, although REST principles contemplate mechanisms that could help circumvent these issues, e.g., the use of *Hypermedia as the Engine of Application State* (HATEOAS), our previous research revealed that REST principles are seldom strictly followed [4]. Similarly, although there have been a number of languages and formalisms suggested for explicitly describing Web APIs very few API providers actually use them. Thus, while providing support for discovering and invoking Web services and their operations can directly be solved by locating and parsing WSDL documents, discovering Web APIs is far more complex a task that is yet to be adequately addressed.

Driven by the aforementioned observations in our ongoing work on iServe [3], a public platform for service publication and discovery, we are approaching the discovery of Web APIs as a three steps activity, whereby we first carry out a targeted crawling of HTML Web pages providing the technical documentation of Web APIs, we subsequently analyse the pages obtained to detect and extract relevant technical information, e.g., operation names, URLs, etc, and we ultimately provide search functionality over the extracted information. In earlier research we have reported our approach and the results obtained in the first step [5]. In this paper we focus on the second step which is in charge of processing the obtained Web pages to extract as much information as possible that shall serve as a basis for advanced Web API discovery. In particular, we present two novel algorithms for supporting the extraction of key information from Web pages documenting Web APIs providing both Remote Procedure Call (RPC) and RESTful interfaces. The algorithms combine state of the art Information Extraction and Natural Language Processing (NLP) techniques to extract features such as operation names, operation descriptions or URI templates. Our algorithms have been evaluated against a dataset containing 40 highly heterogeneous Web APIs documentation pages covering both RPC and RESTful APIs. The evaluation highlights the superior performance of our approach over state of the art Information Extraction algorithms, reaching a precision of 89% when detecting operations and their corresponding information for RPC Web APIs, and 93.4% when detecting blocks describing methods[3] for RESTful Web APIs.

The remainder of this paper is organised as follows. In Section 2 we cover the related work and provide additional background information relevant for the topic at hand. We next explain the approach we have followed and describe the algorithms we have developed. In Section 4 we present the experiments we have carried out and discuss the evaluation results, and finally in Section 5 we present the main conclusions we have drawn and introduce lines for future research.

[3] In Section 3 we explain what we understand by method block in this context and why it is relevant.

2 Background and Related Work

2.1 Service Discovery

Service discovery has been the subject of much research and development. The most renown work is perhaps Universal Description Discovery and Integration (UDDI) [2], while nowadays Seekda[4] provides the largest public index with about 29,000 WSDL Web services. Research on semantic Web services has generated a number of ontologies, semantic discovery engines, and further supporting infrastructure over the years, see [6] for an extensive survey. Despite these advances, however, the majority of these initiatives are predicated upon the use of WSDL Web services, which have turned out not to be prevalent on the Web where Web APIs are increasingly favoured [3].

A fundamental characteristic of Web APIs is the lack of standardisation both concerning their implementation as well as concerning their publication and documentation. An analysis we carried out manually highlighted the heterogeneity existing both in terms of the type of interfaces provided with only 32% apparently RESTful, and the remaining 68% being either purely RPC or hybrid [4]. Additionally, although a few providers do follow REST principles like HATEOAS that enable to some extent the automated discovery and invocation of these services, most do not and provide instead weakly semistructured Web pages documenting the APIs with a highly variable degree of detail (e.g., the HTTP method used is not always indicated, etc) [4]. Furthermore, while the documentation of a single Web API typically follows a certain pattern locally, the structure adopted by different APIs documentation pages is highly heterogeneous which prevents the direct application of general pattern-based solutions. To address these issues, researchers have proposed a number of languages and formalisms for describing APIs, e.g., WADL [7] or for annotating the Web pages documenting them such as SA-REST [8] and hRESTS/MicroWSMO [9]. However, their adoption remains minimal outside academic environments.

As a consequence, there has not been much progress on supporting the automated discovery of Web APIs. The main means used nowadays by developers for locating Web APIs are the use of traditional search engines like Google or searching through dedicated and registries. The most popular directory of Web APIs is ProgrammableWeb[5] which, as of June 2012, lists about 6,200 APIs and provides rather simple search mechanisms based on keywords, tags, or a simple prefixed categorisation. Based on the data provided by ProgrammableWeb, APIHut [10] increases the accuracy of keyword-based search of APIs compared to ProgrammableWeb or plain Google search. Unfortunately, on the one hand, general purpose Web search engines are not optimised for this type of activity and often mix relevant pages documenting Web APIs with general pages, e.g., blogs. On the other hand, current registries provide more focussed information, but still present a number of issues. First and foremost, more often than not, these registries contain out of date information or even provide incorrect links to APIs documentation pages. Indeed, the manual nature of the data acquisition

[4] `http://webservices.seekda.com/`
[5] `http://www.programmableweb.com`

in APIs registries aggravates these problems as new APIs appear, disappear or change. Secondly, the fact that the data listed is often not that accurate and rather coarse grained hampers significantly the development of advanced search functionality since automated algorithms are mislead and miss relevant information such as the operations provided.

Therefore, despite the increasing relevance of Web APIs, there is hardly any system available nowadays that is able to adequately support their discovery. The main obstacles in this regard concern first of all, the automated location of Web APIs, and subsequently, the gathering, interpretation and extraction of relevant information concerning these APIs which is the main focus of this paper. In this regard, to the best of our knowledge besides our own work we are only aware of another initiative which has carried out some initial steps in a similar direction [11]. However, at the time of this writing, details on the experiments and the results obtained are hardly available and, according to the authors, require further refinement.

2.2 Web Page Analysis and Information Extraction

As previously argued, obtaining the necessary information for discovering or invoking the vast majority of Web APIs nowadays requires interpreting highly heterogeneous HTML pages providing documentation for developers. Although, to the best of our knowledge, no other approaches to automating the extraction of information from Web APIs documentation have been devised so far, considerable effort has been devoted to extracting information from Web pages which is relevant to this work.

Tag-based Segmentation approaches are used to analyse the DOM tree of Web page and automatically divide it into subtrees in order to eventually extract information. The essence of this approach relies on the observation that useful information is usually wrapped into so-called *important blocks*. These techniques usually rely on specific HTML tags, e.g., `<table>` in the case of [12], as block separators. This approach is, however, not performant when dealing with heterogeneous cases where various tags are used as separators. To address this, proposals like [13] contemplate a wider range of tags e.g., `<tr>`, `<hr>` .

Template-based Segmentation techniques rely on the fact that, for repetitive information Web pages often use a recurring structure to capture information [15]. These approaches exploit templates provided either by humans or derived automatically by machine learning techniques in order to extract information from the Web pages. Although, these techniques are performant when dealing with Web pages that share (at least partly) a common structure, in cases where the heterogeneity of the pages is very high the performance is considerably affected. As we shall see, although it is possible to exploit local patterns within a Web API documentation, currently diverse Web APIs use highly diverging structured which prevents us from applying successfully this approach.

Vision-based Segmentation is another popular family of approaches that exploits the technique the “visual" boundaries between blocks of content as a means to segment Web pages into blocks of information. In [16] the authors present the popular Vision-based Page Segmentation Algorithm (VIPS) which

views Web pages as (a collection of) images and traverses the DOM tree detecting the “visual" boundaries between blocks. VIPS's granularity of segmentation is based on a *predefined degree of coherence* (PDoC) which acts as a lower bound for the computed *degree of coherence* (DoC) of the identified page blocks. A number of researchers have used VIPS as the underlying page segmentation algorithm, see for instance [17–19]. One weakness of these approaches is due to the fact that the visual boundaries of Web pages are often not clear cut. Additionally, solutions based on VIPS sometimes face granularity problems, which may lead to either too granular segmentations when the PDoC is high, or too coarse grained segmentations when the PDoC is low.

3 Information Extraction from Web API Documentation

Given a Web page documenting a Web API our approach to extracting these features consists on two main steps. On a first step, as is common practice when processing Web pages, we pre-process the Web page to i) circumvent any issues with incorrect HTML pages[6], and ii) to remove scripts and images which will not be taken into account for feature extraction. The second step, takes care of processing and analysing the cleaned Web pages to extract the main technical features of the Web APIs such as the operation names, URI templates, etc.

As we introduced earlier, currently two main styles for implementing Web APIs coexist on the Web, i.e., REST and RPC. Both styles embed a number of architectural and design decisions that condition the way service interfaces are defined and documented, and which, as we shall see, condition the kinds of techniques we can successfully apply to extract information. In the remainder of this section we describe the approach and algorithms we have devised for extracting information from each of these types of interfaces.

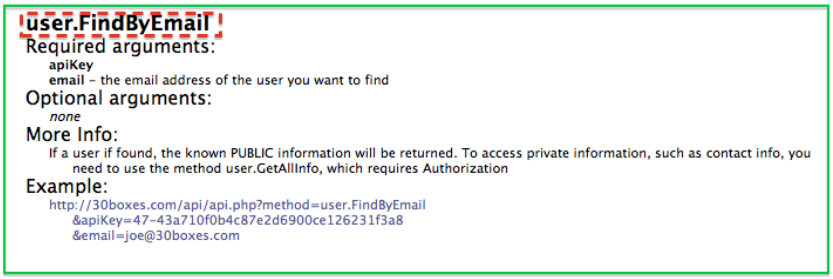
3.1 Information Extraction from RPC-style Web APIs

RPC is a communication style that is centred on the notion of operations or procedures which essentially define the actions that remote components can trigger. Web APIs adopting this style of communication offer programmatic access to the data and functionality hosted on the underlying Web site by means of an arbitrary number of operations. Additionally, the documentation sometimes indicates further details such as the invocation endpoint (a URL) or the HTTP method to be used. Figure 1 shows two examples with the relevant blocks of information extracted from RPC-style Web APIs whereby dashed border rectangles represent the operation identifiers and plain border rectangles represent the information blocks that provide further details, e.g., parameters, and describe what the operations do in natural language. Even though the blocks are both taken from RPC-style Web APIs documentation, the underlying structure and their visual appearance differ significantly, providing an example of the heterogeneity among Web APIs documentation.

In extracting relevant information from RPC-style Web APIs the essence of our approach in centred on the detection of blocks providing information about

[6] We used the Jericho Parser `http://jericho.htmlparser.net/docs/index.html`

(a) Block extracted from http://www.benchmarkemail.com/API/Library



(b) Block extracted from http://30boxes.com/api/

Fig. 1: Examples of block content in RPC-style Web APIs.

the different operations. This detection relies on two main characteristics we have observed while analysing a plethora of Web APIs documentation. First and foremost, we exploit the fact that operation identifiers typically use words in CamelCase[7] notation whereby the first part is a verb and the rest a noun, e.g., *getArtist*. Secondly, since documentation pages are designed for humans, a single Web API documentation often uses some recurrent visual clue to provide a distinct visual appearance to the different elements, e.g., operation names are all within an `<H3>` tag, the description within a `<p>` tag, etc. Thus, although structural patterns across Web APIs documentations cannot be exploited given the existing heterogeneity we previously highlighted, we do utilise the fact that there typically exist local patterns within the same API documentation.

Algorithm 1 details how we extract operation description blocks from RPC-style Web APIs. In a first step we extract all the CamelCase words we can find from the page and track the immediate tag they belong to. In this step, see line 4, we only keep track of the CamelCase words that are composed of a verb and a noun, which, as we shall see in Section 4, is a significant improvement. To this end we use the Log-linear Part-Of-Speech Tagger implemented by the Stanford NLP group[8]. This tagger reads text in english and assigns a part of speech to each word such as verb, noun, and adjective. Doing so allows us to distinguish *getArtist* which denotes potentially an operation identifier as it is made of the verb *get* and the noun *Artist*, from *pageNumber* which is composed of two nouns, *page* and *Number*, and should hence be discarded.

[7] `http://en.wikipedia.org/wiki/CamelCase`
[8] `http://nlp.stanford.edu/software/tagger.shtml`

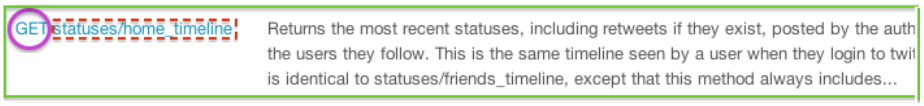
```
   input  : Source page of the Web API
   output: Operation description blocks
 1 foreach tagContent in page source do
 2 |   camelCaseList ← tagContent.getCamelCaseWords();
 3 |   foreach ccWord in camelCaseList do
 4 |   |   if isOperation(ccWord) then
 5 |   |   |   tagMap.Add(Pair<tagName, ccWord >);
 6 |   |   end
 7 |   end
 8 end
 9 electedTag ← getMostPopularTag(tagMap);
10 operationList ← getOperations(tagMap, electedTag);
11 operationMap ← ∅;
12 foreach tagContent in page source do
13 |   operations ← tagContent.getCamelCaseWords() ∩ operationList;
14 |   foreach ccWord in operations do
15 |   |   if getTag(ccWord) == electedTag then
16 |   |   |   if getTag(other words in operations) ≠ electedTag then
17 |   |   |   |   operationMap.Add(ccWord → tagContent);
18 |   |   |   end
19 |   |   end
20 |   end
21 end
22 return operationMap;
```

Algorithm 1: Block detection for RPC-style Web APIs.

In a second step, we find out the HTML tag that has most commonly been used for the operation identifiers candidates found in the first step. This tag, which we refer to as the *elected tag*, is obtained by function `getMostPopularTag` in line 9. Next, out of all the operation identifiers candidates, we only retain the operation identifiers candidates that were within the elected tags. Finally, the tag scope for each of the operations found is eventually used in order to segment the page into blocks of additional information related to a single operations (e.g., operation description, parameters, etc), see Figure1 where the outer rectangle delimits a single block and the dashed inner rectangle highlights the operation indentifier detected.

3.2 Information Extraction from RESTful Web APIs

Web APIs that conform to REST principles [1] are characterised by a communication model whereby requests and responses are built around the transfer of representations of resources and a prefixed set of operations one can carry over these. The documentation of RESTful Web APIs is thus centred on the notion of resource and the parameters required for identifying these resources as well as on the actual operations allowed over the resources which in the case of Web APIs is indicated by the HTTP method to be used. Figure 2 shows two examples of blocks of information extracted from RESTful Web APIs whereby the inner dashed rectangles represent the resources URI (templates), circles identify the HTTP method to be used and the outer rectangles capture the entire segment of information.



(a) Block extracted from https://dev.twitter.com/docs/api

URL	http://del.icio.us/api/[username]/bookmarks/[hash]
Method	GET
Returns	200 OK & XML (delicious/bookmark+xml)
	401 Unauthorized
	404 Not Found

(b) Block extracted from http://www.peej.co.uk/articles/restfully-delicious.html

Fig. 2: Examples of block content in RESTful Web APIs.

Given the characteristics of RESTful interfaces, the main goal in this case is to detect and extract the resources, their URIs (templates) along with the HTTP method to be used and the corresponding description (see plain border rectangles in Figure 2). As opposed to the previous case, naming conventions here are hardly usable since there is no usual criterion for defining resource identifiers. Instead, the most characteristic elements in this case are URI templates and HTTP methods although they occur in several places throughout the documentation (e.g., one may `GET` and `POST` a resource) which hampers significantly the segmentation of the Web page into coherent sets of resources and operations. In order to deal with this, we exploit in this case the fact that within a single API developers often adopt the same repetitive pattern for documenting each resource and the operations available.

In particular, our algorithm, see Algorithm 2, first analyses the structure of the API documentation in order to segment the Web page into similar structural blocks and keeps those blocks that are believed to provide technical documentation since they contain URIs and, whenever available, HTTP methods, e.g., `GET, POST, PUT, DELETE`, see line 1. In a second step, see lines 2 onwards, the algorithm computes the similarity of the blocks detected in order to retain those that appear to exhibit the same structure. Once the blocks have been filtered, we extract the internal features contained, e.g., URIs, HTTP methods, etc, and treat them as a coherent set of information related to a given resource and the operations it offers.

To compute the similarity of the blocks detected the algorithm exploits the notions of **entropy** and **node internal structure**. Entropy calculation is used to quantify strong local patterns exhibited in a page segment, whereby a high entropy indicates important disorder and a low entropy indicating strong similarity.

```
   input  : Source page of the Web API
   output: URI description blocks
 1 blockPattern ← detectBlockPattern();
 2 currentEntropy ← 1; oldEntropy ← 2;
 3 electedDescriptionMap ← ∅; descriptionMap ← ∅;
 4 while currentEntropy < oldEntropy do
 5 |   descriptionMap ← ∅;
 6 |   oldEntropy ← currentEntropy;
 7 |   foreach tagContent in page source do
 8 |   |   tagStructure ← getStructure(tagName);
 9 |   |   if tagStructure starts with blockPattern then
10 |   |   |   while not tagContent contains a URI do
11 |   |   |   |   Tag ← parentTag; Update tagContent;
12 |   |   |   end
13 |   |   |   descriptionMap.Add(tagStructure → tagContent)
14 |   |   end
15 |   end
16 |   structureList ← descriptionMap.keySet();
17 |   currentEntropy ← getEntropy(structureList);
18 |   if currentEntropy < oldEntropy then
19 |   |   electedDescriptionMap ← descriptionMap;
20 |   end
21 end
22 return electedDescriptionMap;
```

Algorithm 2: Block detection for RESTful Web APIs.

The main concepts underlying the notion of entropy in our approach are as follows. Given a set of nodes in the DOM tree structure of a Web page, f_i denotes the frequency with which the node name i appears and p_i the probability of node i.
p_i is hence calculated via the formula:

$$p_i = \frac{f_i}{n} \tag{1}$$

where, n is the total number of nodes. Using equation (1), the entropy of node v is defined as:

$$E(v) = -\sum_{i \in V} \frac{p_i log(p_i)}{log(|V|)} \tag{2}$$

where, V is the set of unique node names that appear in the subtree of the DOM rooted at node v. We can easily extend this concept in calculating the entropy of a list of elements.

We consider the **internal structure** of a node A as being the concatenation of the node name with all of its children traversed top-down left-right in the subtree rooted at A, e.g if we consider part of the source code being `<div><a>title</a><p>body of the div <br></div>`, the internal structure

of the `<div>` node is `<div><a><p><br>`.
In order to detect repetitive block patterns within the structure of the HTML page, see function `detectBlockPattern` on line 1, we base our approach on the ideas behind the "Repetition-based Web Page Segmentation" (REPS) algorithm [19] which proposes a flexible approach for recognising repetitive tag patterns in the DOM tree structure of a Web page, and the approach proposed in [20] by Gujjar Vineel who proposed a DOM tree mining approach for page segmentation where he introduced the concept of *page segment nodes* that would characterise nodes that divide the DOM Tree into logical blocks.

In essence, we use the approach in [20] to segment the Web page into logical blocks (e.g., header, footer, left column, right column, content), then in order to find repetitive patterns within the DOM we exploit the approach proposed by REPS and apply it on the each of the logical blocks. This way, we can keep track of logical blocks that turn out to have repetitive patterns with URI within. Eventually, we keep the most repeated pattern and refer to it as *blockPattern*. Therefore, we can exploit the patterns detected to divide the logical block containing the *blockPattern* into sub items that should in principle represent the blocks describing each resource and their operations.

4 Evaluation

In order to evaluate our approach, we manually generated a dataset based on the documentation of 40 Web APIs. The APIs selected were such that 20 had an RPC interface and the other 20 a RESTful interface. APIs) and documentations within each of these were had a significantly different structure (e.g., documentation structured on the basis of tables, HTML headers, flat documentations, etc). Each of the selected Web API documents were manually analysed in order to extract the information captured within them. The resulting dataset contains 355 URIs for resources of RESTful Web APIs and 515 operations for RPC-style Web APIs. For RPC-sytle Web APIs, we stored for each encountered operation: i) the URL of the Web API documentation, ii) the operation identifier, and iii) the description block that also contains the (required) parameters, and other information regarding what the operation does. For RESTful Web APIs we captured for each resource: i) the URL of the Web API documentation, ii) the URI (template), iii) the HTTP method, and iv) the description block. The interested reader can find concrete details about our dataset as well as subsequent evaluations of our algorithms in the website of iServe[9].

On the basis of the dataset generated, we evaluated the algorithms developed. Notably, we applied the algorithms over each Web API documentation to automatically extract the main technical features of the Web APIs. We then computed the overall precision and recall [10] by contrasting the automatically extracted information with the one we manually extracted and recorded in the dataset.

[9] `http://iserve.kmi.open.ac.uk/datasets/apis-information-extraction.html`
[10] `http://en.wikipedia.org/wiki/Precision_and_recall`

4.1 Evaluation Results for RPC-style Web APIs

In order to evaluate our algorithm for extracting information from RPC-style Web APIs documentation we tested our own algorithm against the dataset and compared it with other 2 techniques that we used as baseline. Given that no other information extraction techniques have been devised for this purpose, we compared the different techniques with respect to their accuracy for detecting operation identifiers. The subsequent extraction of the operation description block was in all cases based on the same common procedure we devised for Algorithm 1 taking the detected operation identifiers as a starting point. The first technique–T_1–was solely based on the detection of CamelCase words, whereas T_2 was based on the combination of CamelCase word detection and the use of Part-of-Speech tagging to filter out the operation identifiers that were not composed of a verb and a noun. Finally, our algorithm which extends T_2 with the notion of *elected tag*, see Algorithm 1, is referred to in our evaluation as T_3. Table 1 provides a summary of the results for the three techniques. Regarding the results obtained for technique T_3, at the time we were experimenting this approach, page 20[11] could not be tested as it was no longer documenting a Web API. This shows how fast references to Web APIs documentation are changing which motivates further the work presented herein.

Overall, T_1 obtained an average precision of 54.68% and an average recall of 94.55%. These results confirm the fact that most operation identifiers are CamelCase words (see the high recall), but also the fact that there are many CamelCase words that are not necessarily operation identifiers (hence the low precision). A detailed look at the extracted operation names for instance included words like USgov, pageNumber and LinkedIn. The second technique, which exploits Part-Of-Speech tagging, exhibited a significant precision improvement over T_1 of about 20 percentage points, with a minor decrease in recall of about 3.5 percentage points. These results thus highlight the considerable benefits that are obtained by using Part-Of-Speech tagging, but still the overall precision, 75%, remains rather low. Finally, our algorithm exhibited the best precision with 92.71%, i.e., an increase of 38 percentage points over T_1 and 17 over T_2. Similarly, the overall recall exhibited by our algorithm, although minimally worse than the one for the other techniques, remains very good with 90.59%. This result thus highlights the fact that using the notion of *elected tag* which exploits the recurrent use of tags for structuring documents is a good means to fine tune the extraction of information from RPC-style Web APIs documentation. The results illustrate that, by combining CamelCase word detection, Part-of-Speech tagging and the notion of *elected tag*, we succeeded in getting better precision and recall as well as on providing a more stable algorithm.

Based on the operations identifiers detected, we also evaluated the performance of our algorithm for detecting the entire operation description blocks, see lines 11 to 21 in Algorithm 1. The results were very similar with a minor decrease both in precision and recall, notably for T_3 we obtained a good performance with a precision of 89.09%, and a recall of 84.56%. This slight performance decrease

[11] `http://api.conceptshare.com/API/API_V2.asmx`

Table 1: Operation detection from RPC-style Web APIs.

	Tech	Pages																				Avg
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	
Precision	T_1	46.8	31.2	**100**	71.4	37.5	58.3	80	66.6	62.5	**100**	65.9	78	27.6	16.6	26	12.	9.5	66.2	90.4	46.1	54.68%
	T_2	68.1	62.5	**100**	**92.6**	72.7	87.5	**100**	83.3	83.3	**100**	90.6	92.9	44.5	31.8	46.2	26.2	50	88.6	95	**85.7**	75.08%
	T_3	**68.2**	**100**	96	88.9	**93.3**	**100**	83.3	**100**	**100**	96.9	**97.5**	**97**	**63.6**	**81.8**	**100**	**100**	**94.7**	**100**	**100**	-	**92.71%**

	Tech	Pages																				Avg
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	
Recall	T_1	**68.2**	83.3	98.9	**100**	90	93.3	**100**	**100**	100	**100**	86.1	**100**	94.2	**100**	**100**	94.4	**100**	82.4	100	**100**	**94.55%**
	T_2	**68.2**	83.3	76.3	**100**	80	93.3	**100**	83.3	100	**100**	80.5	**100**	92.9	**100**	92.3	88.9	**100**	81.6	100	**100**	91.04%
	T_3	57.7	83.3	**100**	80	**93.3**	**100**	83.3	**100**	100	77.5	**100**	92.9	**100**	69.2	90	**100**	93.9	**100**	100	-	90.59%

when extracting operation description blocks is due to the fact that in a few cases the Web pages do not strictly partition operations' details into different blocks. In these cases, the next level up in the DOM structure with respect to the tag holding the operation identifier contains several operation description blocks rather than just the one sought.

4.2 Evaluation Results for RESTful Web APIs

Like in the previous case we evaluated our algorithm for RESTful APIs and compared it with 2 techniques that we used as baseline. Given that no other information extraction techniques have been devised for this purpose, the comparison was based in this case on the application of diverse heuristics for detecting block patterns since this is the key step prior to extracting the relevant information about resources and how to interact with them, see line 1 of Algorithm 2. The first technique–T_4–was based on the direct application of the approach in [20] for block pattern detection. T_5, on the other hand was based on the use of REPS [19] exclusively. Finally, our algorithm which combines RESP and the approach in [20] to detect block patterns, see Algorithm 2, is referred to in our evaluation as T_6. The subsequent processing exploiting the notion of entropy to select the most promising blocks was in all cases based on our own algorithm.

Table 2 summarises the results obtained in terms of precision and recall when applying T_4, T_5 and T_6 on the RESTful APIs recorded in our dataset. The results obtained show that our approach outperforms state of the art techniques (i.e. T_4 and T_5) in both terms of precision and recall. Notably, we obtain an average precision of 93.4% and an average recall of 83.53% whereas for T_4 and T_5 obtain respectively average precisions of 88.91% and 92.52% and average recalls of 69.18% and 72.46%. The results illustrate that, in most cases, by combining ideas from [20] and [19] we succeeded in getting better precision and recall as well as on providing a more stable algorithm.

Table 2: Block detection from RESTful Web APIs.

	Tech	Pages																				Avg
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	
Precision	T_4	100	**71.4**	**100**	100	100	50	100	**100**	100	23.7	100	100	100	100	10.8	100	100	**100**	100	100	88.91%
	T_5	100	46	**100**	100	100	**100**	100	80	100	**100**	100	100	100	100	**100**	**100**	100	9.4	100	100	92.52%
	T_6	100	27	93.8	100	100	**100**	100	85.7	100	48.2	100	100	100	100	**100**	100	100	**100**	100	100	**93.4%**

	Tech	Pages																				Avg
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	
Recall	T_4	100	100	3	80	**100**	100	17.4	84.6	65	**100**	28.2	33.1	31.7	100	**100**	100	53.6	100	100	25.2	69.18%
	T_5	100	100	18.9	80	39.15	100	**17.6**	**92.3**	65	49.9	37	**45.5**	48.6	100	85.7	100	**96.4**	100	100	**92.6**	72.46%
	T_6	100	100	**100**	80	**100**	100	10.8	**92.3**	65	**100**	**44.3**	33.8	**100**	100	85.7	100	**96.4**	100	100	**92.6**	**83.53%**

5 Conclusion

Despite the availability of a number of best practices, e.g., REST principles, and a plethora of software components and technologies, discovering and exploiting Web APIs requires a significant amount of manual labour. Notably developers need to devote efforts to interacting with general purpose search engines, filtering a considerable number of irrelevant results, browsing some of the results obtained and eventually reading and interpreting the Web pages documenting the technical details of the APIs in order to develop custom tailored clients.

In an attempt to provide further automation while carrying out these activities in this paper we have presented an approach for extracting automatically relevant technical information from Web pages documenting APIs. Notably, we have devised two algorithms that automatically and accurately extract Web APIs features such as operation names, operation descriptions or URI templates from Web pages documenting APIs adopting both RPC-style interfaces or RESTful interfaces. We have manually generated a detailed evaluation dataset which we have used to test and compare our approach. The evaluation results show that we achieved a high precision and recall–around 90% in both cases–outperforming state of the art information extraction algorithms.

In our future work we plan to apply our algorithms over a large scale Web crawl in order to develop a fully-fledged Web APIs search engine based on the features extracted. This work is expected to eventually support a greater search accuracy as well as finer grain discovery support than what is currently available nowadays.

References

1. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
2. Erl, T.: SOA Principles of Service Design. The Prentice Hall Service-Oriented Computing Series. Prentice Hall (July 2007)

3. Pedrinaci, C., Domingue, J.: Toward the Next Wave of Services: Linked Services for the Web of Data. Journal of Universal Computer Science **16**(13) (2010) 1694–1719
4. Maleshkova, M., Pedrinaci, C., Domingue, J.: Investigating Web APIs on the World Wide Web. In: European Conference on Web Services (ECOWS), Ayia Napa, Cyprus (2010)
5. Lin, C., He, Y., Pedrinaci, C., Domingue, J.: Feature lda: a supervised topic model for automatic detection of web api documentations from the web. In: The 11th International Semantic Web Conference (ISWC), Boston, USA (2012)
6. Pedrinaci, C., Domingue, J., Sheth, A.: Semantic Web Services. In: Handbook on Semantic Web Technologies. Volume Semantic Web Applications. Springer (2010)
7. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly Media, Inc. (May 2007)
8. Sheth, A., Gomadam, K., Lathem, J.: SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups. Internet Computing, IEEE **11**(6) (Nov 2007) 91 – 94
9. Kopecky, J., Vitvar, T., Pedrinaci, C., Maleshkova, M.: RESTful Services with Lightweight Machine-readable Descriptions and Semantic Annotations. In: REST: From Research to Practice. Springer (2011)
10. Gomadam, K., Ranabahu, A., Nagarajan, M., Sheth, A.P., Verma, K.: A faceted classification based approach to search and rank web apis. In: ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services, Washington, DC, USA, IEEE Computer Society (2008) 177–184
11. Steinmetz, N., Lausen, H., Brunner, M.: Web service search on large scale. In: Proceedings of the 7th International Joint Conference on Service-Oriented Computing. ICSOC-ServiceWave '09, Berlin, Heidelberg, Springer-Verlag (2009) 437–444
12. Lin, S., Ho, J.: Discovering informative content blocks from Web documents. Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining (2002) 588–593
13. Debnath, S., Mitra, P., Pal, N.: Automatic Identification of Informative Sections of Web Pages. IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING **17**(9) (2005)
14. Chakrabarti, D., Kumar, R., Punera, K.: Page-level template detection via isotonic smoothing. Proceedings of the 16th international conference on World Wide Web (2007) 61–70
15. Hammer, J., Garcia-Molina, H., Cho, J., Aranha, R., Crespo, A.: Extracting Semistructured Information from the Web. In Proceedings of the Workshop on Management of Semistructured Data (May 1997)
16. Cai, D., Yu, S., Wen, J.: Vips: a visionbased page segmentation algorithm. Technical Report MSR-TR-2003-79, Microsoft Research (2003)
17. Liu, Y., Wang, Q., Wang, Q., Liu, Y., Wei, L.: An Adaptive Scoring Method for Block Importance Learning. In: Web Intelligence, 2006. WI 2006. IEEE/WIC/ACM International Conference on. (2006) 761–764
18. Wan, X., Yang, J., Xiao, J.: Block-based similarity search on the Web using manifold-ranking. In: Semantic Web: Research and Applications, Proceedings, Peking Univ, Inst Comp Sci & Technol, Beijing 100871, Peoples R China (2006) 60–71
19. Kang, J., Yang, J., Choi, J.: Repetition-based Web Page Segmentation by Detecting Tag Patterns for Small-Screen Devices. IEEE Transaction on Consumer Electronics **56**(2) (May 2010)
20. Vineel, G.: Web page DOM node characterization and its application to page segmentation. In: Internet Multimedia Services Architecture and Applications (IMSAA), 2009 IEEE International Conference on. (2009) 1–6