ACADÉMIE DE BORDEAUX
UNIVERSITÉ DE **B**ORDEAUX
Sciences et **T**echnologies

THÈSE

Présentée au Laboratoire Bordelais de Recherche en Informatique pour obtenir le grade de Docteur de l'Université de Bordeaux

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Mathématiques et Informatique**

Web application development with third-party components

par

Hanyang CAO

Soutenue le 5 Février 2019, devant le jury composé de :

Directeur de thèse
Xavier BLANC, Professeur ..... Université de Bordeaux, France

Rapporteurs
Olivier BARAIS, Professeur ..... Université de Rennes 1, France
Romain ROUVOY, Professeur ..... Université de Lille, France

Examinateurs
Li ZHANG, Professeur ..... Beihang University, Chine
David AUBER, Professeur ..... Université de Bordeaux, France

Abstract

Web applications are highly popular and using some of them (e.g., Facebook, Google) is becoming part of our lives. Developers are eager to create various web applications to meet people's increasing demands. To build a web application, developers need to know some basic programming technologies. Moreover, they prefer to use some third-party components (such as server-side libraries, client-side libraries, REST services) in the web applications. By including those components, they could benefit from maintainability, reusability, readability, and efficiency. In this thesis, we propose to help developers to use third-party components when they create web applications. We present three impediments when developers using the third-party components: What are the best JavaScript libraries to use? How to get the standard specifications of REST services? How to adapt to the data changes of REST services? As such, we present three approaches to solve these problems. Those approaches have been validated through several case studies and industrial data. We describe some future work to improve our solutions, and some research problems that our approaches can target.

Keywords: *Web Application, Library Recommendation, REST, JSON, Specification*

Résumé

Les applications Web sont très populaires et l'utilisation de certaines d'entre elles (p. ex. Facebook, Google) fait de plus en plus partie de nos vies. Les développeurs sont impatients de créer diverses applications Web pour répondre à la demande croissante des gens. Pour construire une application Web, les développeurs doivent connaître quelques technologies de programmation de base. De plus, ils préfèrent utiliser certains composants tiers (tels que les bibliothèques côté serveur, côté client, services REST) dans les applications web. En incluant ces composants, ils pourraient bénéficier de la maintenabilité, de la réutilisabilité, de la lisibilité et de l'efficacité. Dans cette thèse, nous proposons d'aider les développeurs à utiliser des composants tiers lorsqu'ils créent des applications web. Nous présentons trois obstacles lorsque les développeurs utilisent les composants tiers: Quelles sont les meilleures bibliothèques JavaScript à utiliser? Comment obtenir les spécifications standard des services REST? Comment s'adapter aux changements de données des services REST? C'est pourquoi nous présentons trois approches pour résoudre ces problèmes. Ces approches ont été validées par plusieurs études de cas et données industrielles. Nous décrivons certains travaux futurs visant à améliorer nos solutions et certains problèmes de recherche que nos approches peuvent cibler.

Mots clés : *Application Web, Recommandation de la bibliothèque, REST, JSON, Spécification*

LaBRI — 351, Cours de la Libération — 33400 Talence — France

Contents

CHAPTER

1 Introduction

This chapter introduces the context, motivations, and contributions of our thesis. This thesis is based on the increasing need for developers to use third-party components (e.g., libraries, REST services) when they create web applications. More precisely we tackle three main problems faced by these developers: 1) Which components to choose since there exists plenty of ones? 2) How to get knowledge of these components? and 3) How to adapt them to better fit their needs? In this chapter we describe these three problems and highlight the underlying challenges. We then present our main contributions. Finally, we conclude with the general structure of the manuscript.

Contents

1.1 Context: Web application development

Web applications have shown their popularity and importance over the last years. To build a web application, developers need to know some fundamental programming technologies, such as HTML, JavaScript, CSS, database system, etc. Meanwhile, they use some third-party components (e.g., libraries, REST services) to improve the time efficiency and achieve their business logic. We will illustrate the context of web application development in the following example.

Running example. Imagine that *Mr. Jobs* decides to build a web application that could display, backup and update users' public opinions (e.g., Tweets, Facebook and Instagram posts). The web application could present and analyze the public influence of those opinions. For instance, the web application could monitor President Donald Trump's tweets and investigate the financial influence of these tweets (e.g., USD/EUR exchange rate fluctuations, crude oil price changes).

In order to achieve his idea, Mr. Jobs prepares the development context of the targeted web application, as shown in the Figure 1.1. He needs to develop two parts of the web application: server and client. The server concentrates on achieving the business logic. The client is the graphical user interface that runs in a web browser. Both sides would benefit from using third-party components. Using a third-party component improves code quality, prevents errors and speeds up productivity [Baldassarre et al., 2005].

For instance Mr. Jobs tends to include various client-side JavaScript libraries (e.g., *jQuery*, *AngularJS*, *Backbone*) in the Client side. Such libraries would help to visualize acquired data and provide a good-looking and user-friendly interface. Furthermore, Mr. Jobs would certainly need to take advantage of third-party REST services to achieve the business logic. For instance, he would call Twitter REST service [1] to get Donald Trump's tweets, and request Xignite service [2] for real-time global currencies.

We call third-party components both libraries (i.e., server-side libraries and client-side libraries) and REST services. As shown in our simple example, Mr. Jobs then relies on the third-party components to accomplish his web application. In the following, we give some more formal definitions and illustrate the problems developers faced when using third-party components.

Web application. A web application is a web-based system that publishes a set of content and functionality to a wide range of end users [Conallen, 1999]. It employs a client-server structure where the client runs in a web browser [Davidson and Coward, 1999; Gellersen and Gaedke, 1999]. Different from the desktop application, web application doesn't need

1. https://developer.twitter.com/en.html
2. https://www.xignite.com/developers

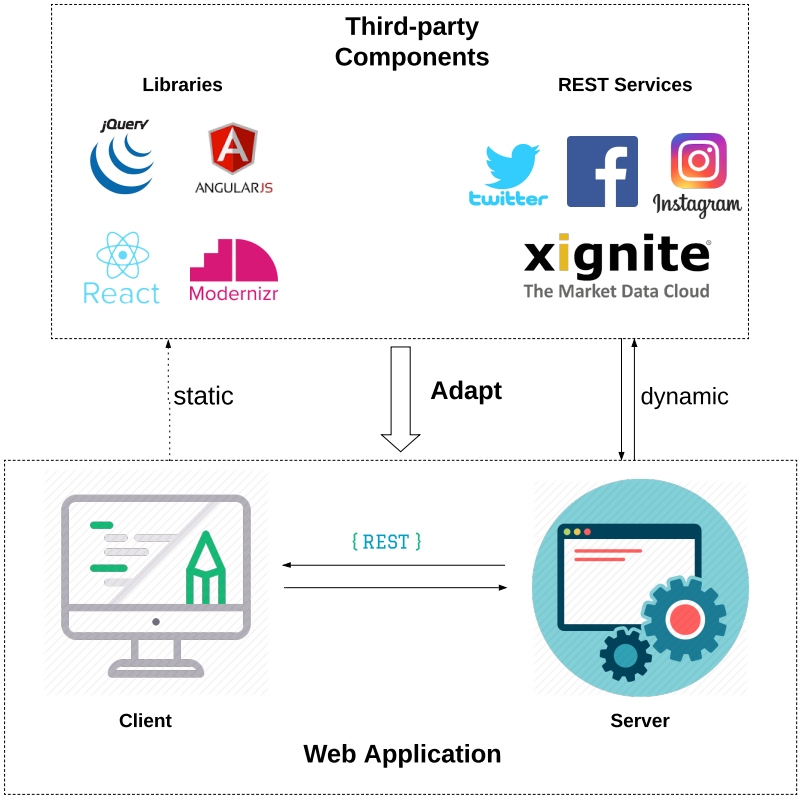


Figure 1.1 – Context of web application development.

to be installed in the operating system, and developers can easily update them without disturbing the users [Per, 2014].

Web application developer. A web application developer is a programmer that builds and maintains web applications. Although there is no black-and-white rule, it can be classified as the server-side and client-side developer. The former concentrates on achieving the business logic while the latter focuses on the data visualization and interaction with users.

third-party components. We call *third-party components* reusable modules or functionalities that are provided by third-party vendors. They are widely used to make the development easier, cheaper and with better quality. The third-party components include libraries and REST services:

— A library is a reusable chunk of source code that helps developers to achieve their business logic. In the context of web application, a library can be either server-side

or client-side. Server-side libraries can be used to connect the database, handle the HTTP communication, automate test, retrieve data from REST service, etc. For instance, *ExpressJS* is a server-side library to set the application, configure the router, and handle requests. Client-side libraries are usually written in JavaScript and can be executed by the browser. They mainly help the developers to handle the HTML DOM elements, cookies, HTTP requests, etc. For example, *jQuery* makes it easier to manipulate an HTML document, select DOM elements, create animations and handle events.

— A REST service gives access to a set of so-called resources [Fielding and Taylor, 2002]. Following the REST principles, the accesses to the resources are all done thanks to HTTP requests, where the verb used by the request defines how the resource is manipulated (`GET` for reading, `PUT` for writing, etc.). For example, Instagram provides a REST service that gives access to the media resources published by its users (pictures, movies, etc.).

When a developer wants to use third-party components he/she encounters the following problems, expressed here as research questions:

RQ1: What are the best third-party components to use? In the running example, let's consider that Mr. Jobs wants to include a client-side library in his project. Since there are more than 50 available candidates (e.g., such as *AngularJS*, *Backbone* or *Polymer*), he then wants to know which one is the best for his needs? He may then ask Google but will get many different answers. Furthermore, if Mr. Jobs wants to include the famous *jQuery* library within its project. By browsing the web, he will then notice that there are two major versions for *jQuery* (1.x.x and 2.x.x) and several minor versions. If he now tries to understand which is the best version, he will end up into StackOverflow with several different and sometimes opposite answers [3].

Recommending third-party components to a potential user has been widely studied in the literature (see details in Section 2.1). It mainly contains two steps: 1) obtain a large dataset of existing third-party components, and 2) sort them according to various criteria (e.g., popularity, similarity). We contribute to that field by focusing on client-side libraries used by famous existing web applications. Such web applications are quite often closed source, which makes them hard to analyze. Hence, our specific research question is *SQ1: What are the best JavaScript libraries to use?*

RQ2: How to get knowledge of third-party components? In our example, Mr. Jobs needs to learn how to communicate with the Twitter REST service to retrieve the real-time tweets. Usually, he will go to the official Twitter HTML documentation cite [4], look up usage instruction of related resources, and then manually write code to handle with the HTTP requests, responses, and authentication. Despite, Mr. Jobs would certainly prefer to have

3. Here is the answer at the time of the 20th of August 2015, http://stackoverflow.com/questions/22289583/what-version-of-jquery-should-i-act ually-use

4. https://developer.twitter.com/en/docs

a machine-readable specification of Twitter service (such as OpenAPI). This specification can be treated as a user guide for computer, to help him deal with the low-level details. However, such Twitter specification currently is not available.

Generating a machine-readable specification from the textual documentation of a third-party component has been studied in the literature (see details in Section 2.1). We contribute to that field by focusing on server-side third-party components and more precisely on REST service. REST services usually provide on-line documentations of their products. However, there are very few machine-readable specifications of them. Developers have then no choice than reading the documentation and comprehend how to use the REST services, which is error-prone. In our thesis, we then aim to answer the following question *SQ2: How to get the standard specifications of REST services?*

RQ3: How to adapt third-party components? In our example, the Twitter service returns a time-line of tweets that changes quite frequently. Mr. Jobs should then call the service periodically to get an update of the time-line, whether the time-line changed or not. The periodically calling may have a high cost in bandwidth and system resources, which is inadequate for services whose data is periodically changing. Mr. Jobs would certainly prefer to adapt the REST service to get notification messages only when the time-line has really changed.

The adaption of existing components is essential. There are mainly two kinds of adaptation: version migration or data adaptation. The version migration is performed when a web application wants to use a new version of a third-party component. In this case, developers need to ensure their API calls are compatible with the target version. A lot of existing work focus on REST service migration [Li et al., 2013; Espinha et al., 2014; Wang et al., 2016] and library migration [Meng et al., 2012; Dig et al., 2008]. The data adaption means that the web applications should efficiently retrieve the frequently updated data. The data adaption only exists in dealing with REST service since other components do not return data. The existing way of communicating with the REST service is to call the service periodically, which is not efficient when data changes frequently and unpredictably. In our thesis, we focus on this kind of adaptation and then aim to answer the following question *SQ3: How to adapt to the data changes of REST services?*

The Table 1.1 present three problems faced by developers of web applications. It further highlights how these problems relate to libraries of REST services. Then it pinpoints the specific cases we choose to focus on in this thesis.

As a summary, in this thesis we aim to help the web application developers by solve three problems (i.e., RQ1, RQ2, RQ3) they encounter when using third-party components. Due to the universality of these problems, we provide concrete solutions for subproblems (i.e., SQ1, SQ2, SQ3). SQ1 is based on investigating the famous web application, which is a unique problem for library and has not been studied before. SQ2 focus on getting the standard specification, which is a unique problem for REST service. SQ3 is also a unique problem that just happens in dealing with REST service. We will illustrate the subproblems in the following section.

Table 1.1 – Research problems about leveraging third-party components in web application development.

	Library	REST Service
What are the best third-party components to use?	Open source web application Famous web application ✓	Open source web application
How to get knowledge of third-party components?	Documentation Source code	Documentation Specification ✓
How to adapt to changes of third-party components?	Version migration	Version migration Data adaption ✓

1.2 Problem Statement

After clarifying the research context and introducing the problems, we present them in more detail in this section. For each of the problems, we present the main motivations, highlight the challenges and discuss the web application developers' expectations.

1.2.1 What are the best JavaScript libraries to use?

Motivation. Using a third-party library provides many benefits as reusing high quality code prevents errors and speeds up productivity[Baldassarre et al., 2005]. However, it comes with the main problem of the choice of the best library to use from a software development perspective [Teyton et al., 2012]. Indeed, there are so many libraries with so many versions that it becomes too complex for a developer to choose which one to include in a software project. This is even more difficult for JavaScript libraries, because of the popularity of JavaScript [5], the outstanding pace of JavaScript libraries production, and the fact that web applications are doomed to evolve at the Internet speed to be used and not to become deprecated [Baskerville et al., 2003].

Challenge. To help developers in this difficult choice, popularity indicators are frequently used with the main hypothesis that most used libraries are the best ones. In order to get those indicators, existing approaches are based on the analysis of the development

5. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

components (source code files or deployment descriptors) and therefore rely on the observation of open source projects that make such components available. This is however not possible with web applications as most of the famous web applications are closed-source applications, and their development components are not available at all. The development components of a commercial web application are only available on-line by browsing it through its root URL.

Expectation. When web application developers choose which JavaScript libraries should be included in their projects, they want to get recommendations of popular JavaScript libraries and related versions. Currently, there is no available solution for identifying the third-party libraries in commercial web applications. We then must offer a solution to recognize JavaScript libraries in the real world and provide popularity indicators to web application developers.

1.2.2 How to get the standard specifications of REST services?

Motivation. The REST architectural style is nowadays very popular and widely adopted by services providers. To use REST services, the web application developers can be helped by two artefacts: a structured REST specification and an HTML documentation. The best practice is to have a REST specification since it can speed up the development process by automatically generating client-side [Fokaefs and Stroulia, 2015], or even service composition [Wagner et al., 2012]. Additionally, a rigorous specification can be used to reach a better quality by inferring parameters dependency constraints [Wu et al., 2013] or performing automating tests production [López et al., 2013] for example.

Challenge. Having a rigorous specification can help developers to accelerate the development process. However, there are few available REST specifications while most of the REST services providers only provide HTML documentations [Danielsen and Jeffrey, 2013]. According to an in-depth analysis of the most 20 popular REST Services [Renzel et al., 2012], only 20% of them provide WSDL [Chinnici et al., 2007] specifications whereas 75% offer no rigorous specification and only plain HTML pages. Such a situation then calls for an automatic transformation of plain HTML documentations into rigorous specifications.

Expectation. When web application developers want to use REST services, they prefer to get REST specifications rather than HTML documentations. Since most of the REST services provide only HTML documentation and rarely present rigorous specification, a solution to transform HTML documentation to REST specification can be provided.

1.2.3 How to adapt to the data changes of REST services?

Motivation. REST APIs together with JSON are commonly used by modern web applications to export their services. However, these services are usually reachable in a pull mode which is not suitable for accessing changing data. The pull mode needs to periodically call the service to check if the data have changed or not. The polling frequency (e.g., five seconds) is predefined by the web application developers, while the data updating moment is random and hard to predict. Furthermore, the pull mode could waste bandwidth by acquiring unchanged data, and increase battery usage for mobile web applications. For the web application developers, they actually need the push mode that will get notification messages only when the data have changed. Hence, turning a service from a pull to a push mode is therefore frequently asked by web application developers.

Challenge. Converting a pull API into a push one obviously requires to make periodical calls to the API but also to create a patch between each successive version of the data. The latter is the most difficult part and this is where existing solutions have some imperfections. Indeed, creating a patch between two documents is a well-known very complex problem [Zhang and Shasha, 1989; Buttler, 2004], which has not been studied yet for JSON documents. A JSON document is a labeled unordered tree that contains arrays (ordered sequences). Creating a patch between two JSON documents may therefore lead to an NP-hard problem depending both on the change operations that are considered (add, remove, move, copy), and on the quality of the created patch (in terms of size).

Expectation. web application developers prefer to acquire data in a push mode that is more adequate for accessing changing data, but very few web applications support it. Therefore a solution to transform a pull mode API into a push one can be provided.

1.3 Contributions

We present our contributions that address the three issues that we have detailed previously. The ultimate goal of this thesis is to solve a few problems when developers use the third-party components in their web applications. To this extent, we propose three main contributions.

1.3.1 What are the best JavaScript libraries to use?

To help web application developers who want to choose which JavaScript (JS) libraries to include in the client side, we overcome the difficulties and exhibit which are the popular JavaScript libraries. We provide an approach that uses both syntactical and dynamical analysis of the on-line resources of the web applications and detects their used JS libraries.

Our approach browses web applications and detects their used JavaScript libraries with the underlying assumption that the libraries they use are most probably the ones that should be used. To get significant results we decide to observe the Alexa global top 100 websites [6]. Based on such observations we can then output trends and give recommendations.

We make the following contributions:

— We provide an automatic and efficient approach that browses web applications and detects their used JavaScript libraries.

— By applying our approach on the 100 most popular websites, we provide statistics of the use of JavaScript libraries.

— We then present how our observations can be used to provide trends and recommendations.

1.3.2 How to get the standard specifications of REST services?

To help the web application developers who want to have REST API specifications for using existing REST APIs, we propose a semi-automated approach *ExtrateREST*, which automates the creation of REST service specifications from existing HTML documentations. Our approach inputs the index page of the HTML documentation of a REST API provider, performs an analysis of all its HTML pages, and generates the corresponding OpenAPI [7] specification. Many specification formats exist, and OpenAPI turns out to be the most popular one, with over 350,000 downloads per month. Furthermore, once an OpenAPI specification exists, translating it into another format such as WADL, for instance, is very easy. For instance, our approach has been used to generate the OpenAPI specification of Instagram (see Figure 2.3) by inputting its HTML documentation (see Figure 2.1). ExtrateREST outputs the four mandatory parts that compose an OpenAPI specification: the base URL, the path templates, the HTTP verbs and the associated formal parameters.

As the main result, we provide:

— A semi-automated approach that generates an OpenAPI specification from the plain HTML documentation of an existing REST service.

— A validation of our prototype and the OpenAPI specifications that are yielded from topmost popular REST services.

— Public directory of OpenAPI specifications for REST services.

6. http://www.alexa.com/topsites
7. https://www.openapis.org/

1.3.3 How to adapt to the data changes of REST services?

We provide a solution to transform pull mode API into a push one. The central of the conversion is to create a patch between each successive version of the JSON document. Existing solutions don't perform well for generating the patch.

To face this issue, we propose a new patch algorithm that is tailored to JSON documents, and that drastically improves the conversion of pull mode APIs into push mode ones. Our algorithm returns a JSON Patch as specified by the JSON Patch RFC [Bryan and Nottingham, 2013]. It therefore handles any changes that can be done on JSON documents, either on their basic properties or on their arrays, and supports simple changes (add, remove) as well as complex ones (move, copy), which allows clients to deeply understand changes that have been done.

We implement our algorithm in JavaScript as it is the most common language used in web applications. We finally provide a very simple prototype showing how it can be used to easily convert a pull mode service into a push mode one.

As the main result, we provide:

— A new JSON patch algorithm that fully complies with the JSON Patch RFC.

— A JavaScript implementation of our algorithm that performs better than the existing ones.

— A prototype framework that can be used to convert a pull service into a push one (see the online demo [8]).

1.4 Thesis Outline

The remainder of this document is organized as follows. We first present in Chapter 2 an overview of the state of the art in the field of third-party components. In Chapter 3, we present an approach to automatically identify JavaScript libraries used in web applications. Then, in Chapter 4, we detail our work on crawling the REST HTML documentation and extracting relevant REST specification. We then present in Chapter 5, our contributions on the generating the JSON Patch for turning a pull REST API into a push one. Finally, we conclude in Chapter 6 by summarizing the contributions and the main perspectives.

8. http://diff-and-patch.netlify.com/

CHAPTER

2 Background

In this chapter, we present the state of the art for each of the three problems addressed in this thesis. Our first problem is "What are the best JavaScript libraries to use?" We then present here related work to third-party components recommendation. The second problem is "How to get the standard specifications of REST services?" We then present the existing literature that is related to the creation of REST specifications. Finally, the third problem is "How to adapt to the data changes of REST services?" We then introduce the two data update styles (pull mode and push mode) and current transformation platforms.

Contents

2.1 What are the best JavaScript libraries to use?

In this section, we present the work done in the fields of third-party component recommendation. As shown in Table 2.1, we summary the existing literature into two aspects: obtain third-party library usage dataset, and sort the libraries according to underlying requirements.

Table 2.1 – Summary table of different approaches dealing with the problem of third-party library recommendation.

		Source			Sort algorithm	
		Open source project	Crowdsourced knowledge	Famous web application	Similarity mining	Popularity indicator
Library	[Thung et al., 2013]	✓			✓	
	[Teyton et al., 2014]	✓			✓	
	[Chen et al., 2016]		✓		✓	
	[Ishio et al., 2016]	✓			N/A	N/A
	[Yu et al., 2017]	✓			✓	
	[Ouni et al., 2017]	✓			✓	
	[Katsuragawa et al., 2018]	✓			✓	
	W3Techs			N/A		✓

Step 1: obtain third-party library usage dataset. In order to build the library usage dataset, we need to identify libraries from various sources such as open source projects, crowdsourced knowledge, or famous web application.

Open source projects. Thung et al. [Thung et al., 2013] describe an approach that supports library recommendation. This approach constructs a dataset of 1008 open source projects from GitHub [1]. Each project in the dataset relies on Maven [2]. The approach then identifies library usage through the Maven configuration files. Some researchers follow

1. http://github.com
2. https://maven.apache.org

this idea and also use Maven to construct their dataset of libraries [Ouni et al., 2017; Katsuragawa et al., 2018].

Teyton et al. [Teyton et al., 2014] analyze a large set of open source software systems to mine their library migrations (replacement of a library by a competing one). They build a dataset of 15168 projects from GitHub. Their dataset is composed of Java project and is not limited to Maven projects.

Yu et al. [Yu et al., 2017] propose a combined approach to identify libraries used in mobile apps. For apps which used Maven to build the system, it analyzes the configuration file (i.e., pom.xml) to identify the libraries. For the rest apps, it analyzes the "import" statements in the source code files to infer the used libraries.

Ishio et al. [Ishio et al., 2016] provide a method to detect third-party components in Java Software Release files. Without the access to the build file of the project, it analyzes the JAVA release jar files (e.g., junit.jar) and succeeds to detect the included components. Given a target Java jar file and a repository of jar files of existing components, it executes a signature-based comparison and selects jar files that are matched.

Crowdsourced knowledge. Chen et al. [Chen et al., 2016; Chen and Xing, 2016] provide an approach that automatically recommends analogical libraries across different programming languages. Instead of obtaining library usage through source code or configuration files, it takes advantages of the crowdsourced knowledge from domain-specific sites (e.g., Stack Overflow[3]). It extracts library usage from tags (e.g., library name, language, os) of Stack Overflow questions.

Famous web application. To the best of our knowledge, there is only one survey, which is done by the W3Techs company, that provides statistics on the library usage among famous web applications[4]. The famous web applications are usually closed source project, and W3Techs does not communicate its underlying identification methodology.

Step 2: sort the libraries according to underlying requirements. For the potential developers, there are two main scenarios where they need library recommendation. The first scenario is when developers want to find similar libraries to replace the ones they use. This scenario happens when developers have an existing project and want to improve its dependencies. In this case, "best" libraries means similar ones with higher capabilities or richer features. The other scenario is when developers want to choose libraries that fit their needs. In this case, we consider that "best" libraries means the most popular ones that fit their needs.

3. https://stackoverflow.com/
4. http://w3techs.com/technologies/overview/javascript_library/all

similarity mining. There are several techniques that are related to similarity mining: Association rule mining [Agrawal et al., 1994], Collaborative Filtering [Terveen and Hill, 2001], Natural Language Processing [Mikolov et al., 2013], Latent Dirichlet Allocation [Blei et al., 2003] and Migration Graph [Teyton et al., 2014]. Many previous studies have proposed approaches to recommend libraries based on these techniques.

To recommend a library, Thung et al. [Thung et al., 2013] use a hybrid approach that combines association rule mining and collaborative filtering. The association rule mining [Agrawal et al., 1994] recommends libraries that are commonly used together. While the collaborative filtering [Terveen and Hill, 2001] recommends similar libraries. Similar to Thung et al., other work also apply this idea to recommend libraries [Ouni et al., 2017; Katsuragawa et al., 2018]

Teyton et al. [Teyton et al., 2014] propose a migration graph method to recommend a given library. It relies on two factors: the technical domain of the library (log, http, database, dom, etc.) and the date of the migration. For each main technical domain, it then presents a migration graph that indicates the best and the worst library depending on when existing migrations have been observed. Thanks to this approach, developers can know the massively adopted library as well as the emerging one.

Chen et al. [Chen et al., 2016; Chen and Xing, 2016] provide an approach to recommends analogical libraries based on a knowledge base of analogical libraries mined from tags of millions of Stack Overflow questions. It takes advantage of nature language processing model [Mikolov et al., 2013] to learn tag embeddings. The aim is to find libraries that are discussed together in the context by developers.

Yu et al. [Yu et al., 2017] propose a hybrid approach AppLibRec which combines Latent Dirichlet Allocation [Blei et al., 2003] and Collaborative Filtering to recommend third-party libraries for mobile applications. The LDA model takes README file (textual description) of a library as input to compute the similarities with other libraries' descriptions.

popularity mining. W3Techs leverages the *wisdom of the crowds* and provides a rank of existing JavaScript libraries according to their popularity. The underlying assumption is that the choice of the majority should be a wise choice. It mines the popular websites according to Alexa ranking [5] and provides total library usage statistics. The popularity of libraries can be used as recommendations for a developer who wants to build a new web application.

Summary. To set up the library usage dataset, existing approaches mainly rely on open source project or other crowdsourced knowledge (as shown in Table 2.1). There are some obstacles to identify the library usage for famous web applications as most of the famous web applications are closed-source applications.

5. https://www.alexa.com/siteinfo

There are two main scenarios related to sort and recommend libraries. Existing approaches focus on the first scenario where developers want to find similar libraries to replace the used ones. They utilize several similarity mining algorithms to find "best" libraries. We want to concentrate on the second scenario where developers want to choose appropriate libraries when they design the web applications. We choose to use library popularity among famous web applications to rank the libraries. The underlying assumption is that the "best" libraries should be the ones that used by the majority of people.

2.2 How to get the standard specifications of REST services?

In this section, we first discuss some concepts related to REST services in detail, including the REST Service, REST API, REST Endpoint, REST API HTML documentation, and REST API specification. We then focus on the existing literature that is related to the creation of REST API specifications. They can be classified into two categories: (semi-)automated approaches, and crowd-sourcing approaches.

2.2.1 REST concepts

REST Service. A REST service gives access to a set of so-called resources [Fielding and Taylor, 2002]. Following the REST principles, the accesses to the resources are all done thanks to HTTP requests, where the verb used by the request defines how the resource is manipulated (`GET` for reading, `PUT` for writing, etc.). For example, Instagram provides a REST service that gives access to the media resources published by its users (pictures, movies, etc.).

REST API. Any REST service has a REST API that is used by client applications to access this service. For instance, Instagram provides its REST API [6] used by many web client, such as AK Stogram that is an Instagram media viewer with many fancy features [7].

REST Endpoint. A REST endpoint describes the access to a resource with a reference URL, a relevant HTTP verb and optionally several parameters and a corresponding response example. For example, the Instagram endpoint `GET /media/media-id` describes how to access *media* resources with a required parameter `ACCESS_TOKEN`. The data returned by a REST service is commonly encoded in JSON.

6. https://www.instagram.com/developer/
7. https://www.4kdownload.com/products/product-stogram

REST API HTML documentation. A REST API HTML documentation describes a REST API using plain HTML files. It is composed of a set of web pages. Among the set of pages, one page is called the *index page*, and is linked directly or indirectly to all the pages of the set. All the pages belong to the same domain (the one of the index page). Furthermore, each page may or may not contain useful information to access the service. Finally, the HTML layouts and vocabulary can be different from a provider to the other.

As an example, the Figure 2.1 shows the index page of the Instagram API HTML documentation. This page directly or indirectly points to a set of 24 web pages that compose the full HTML documentation of the Instagram REST API. Looking more closely, some of these pages are useful as they describe Instagram endpoints. For example, one page describe the *media* resource (see Figure 2.2a). Some pages can be considered to be useless regarding this purpose as they don't describe how to access the service (see Figure 2.2b that gives information about how developers can receive support but does not describe how to access the REST service).

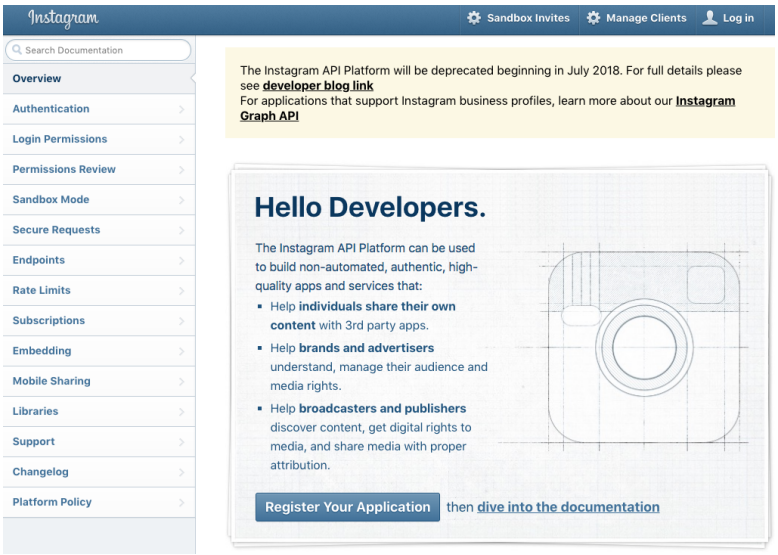
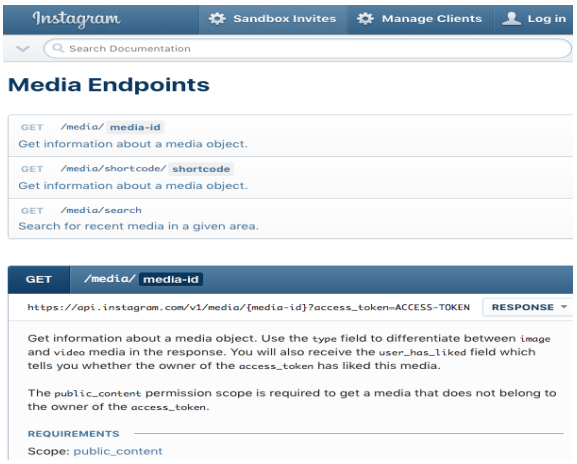


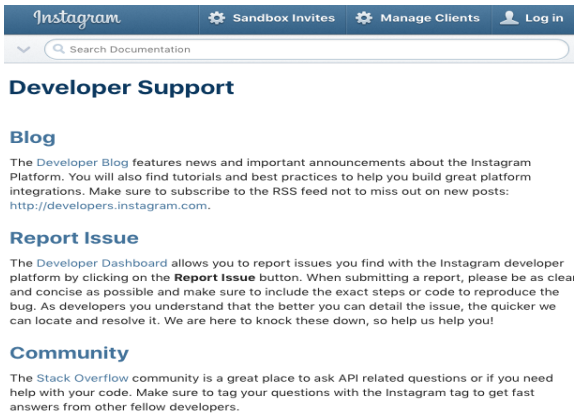
Figure 2.1 – REST API HTML documentation of Instagram.

REST API specification. A REST API specification rigorously defines how to access the resources provided by a REST service. Several papers have focused on providing the standards of machine-readable specifications.

— WADL: Web Application Description Language (WADL) is the *de jure* standard submitted by W3C in 2009 [Hadley, 2009]. WADL uses XML description to model the REST resources and their links. The XML format is machine-readable, but not human-readable. Due to its complexity, WADL doesn't gain the interest of API providers [Ed-douibi et al., 2017; Kopecký et al., 2008].



(a) Media Endpoint screenshot



(b) Developer Support screenshot

Figure 2.2 – Two screenshots in Instagram REST API HTML documentation

— hRESTS: HTML for RESTful Services (hRESTS) is a microformat for web developers to make the main information machine-readable in the HTML [Kopecký et al., 2008]. They add the pre-defined REST services annotations (e.g., HTTP methods, address, output) into the HTML documentation according to hRESTS format. Hence, they transform the human-readable HTML into a machine-readable one.

— RESTdesc: RESTdesc is a semantic web service description [Verborgh et al., 2013] that based on semantic web language Resource Description Framework (RDF) [Klyne and Carroll, 2004]. It is a rule-based format that defines the preconditions, postconditions and request details of the interaction with the REST service. RESTdesc is a simple and lightweight description while still expresses the semantic of REST services.

— WIfL: Web Interface Language (WIfL) defines a set of RDFa annotations that can be injected into the HTML [Danielsen and Jeffrey, 2013]. WIfL provides an interactive console that enables the developers to send the requests and analyze the sample responses directly in the HTML documentation. Moreover, they provide a WIfL tool that can validate the consistency of the REST APIs.

As a summary, even though a set of specification standards has been proposed by researchers, none of them has been widely adopted by REST service providers [Verborgh et al., 2013]. The industry practitioners have also proposed several descriptions, such as RAML [8], API Blueprint [9], and OpenAPI. These API descriptions have a lot in common while OpenAPI seems to be the winner in the "Great API Description Wars" [Tam, 2017], with

8. http://raml.org/
9. https://apiblueprint.org/

over 350,000 downloads per month. Furthermore, translating an OpenAPI specification into another format is easy, existing tools (e.g., Apimatic [10]) already support this need.

OpenAPI. The OpenAPI specification (formerly known as Swagger) is the *de facto* standard that introduced by SmartBear Software. A variety of big technology companies (e.g., Google, Microsoft, IBM) have sponsored OpenAPI, which makes it quickly become the most popular specification description.

At least, it has to describe the following information:

— Base URL: The Base URL is the common prefix of all URLs that give access to the resources.

— Path Templates: The templates describe how the Base URL must be completed to make an URL that does give access to a resource. A path template is a relative path that starts with a leading slash (/).

— Verbs: The verbs list, for each Path Template, the HTTP verbs that are supported by the REST service (GET, PUT, POST, etc.).

— Parameters: The parameters, for each couple of *Path Template* and *Verb*, define the list of formal parameters that are supported by the request. Each parameter must have a name and may have a type (String, Integer or even JSON or XML Schema).

— Response Schema: The Response Schema is the definition of response structure. It is based upon the JSON Schema specification [11], and is compatible with XML. This is an optional part of the specification.

As an example, the Figure 2.3 shows a small extract of the Instagram REST API specification generated by our approach. This extract indicates the base URL of the REST API (https://api.instagram.com/v1), and then indicates that the media resources can be obtained by sending `GET` HTTP requests targeting the *"/media/{media-id}"* template. Further, it indicates that the response will be a `Media` object, and gives the corresponding JSON schema.

2.2.2 Automated or semi-automated approaches

In this section, we present the literature related to the creation of the OpenAPI specification. Automated or semi-automated approaches use a tool that helps the developers to build the specification.

SpyREST. Sohan et al. [Sohan et al., 2017, 2015] provide *SpyREST*, an approach for generating and maintaining up-to-date documentation by using an HTTP proxy server. The

10. https://apimatic.io/transformer
11. http://json-schema.org/

```
{
  "swagger": "2.0",
  "host": "api.instagram.com",
  "schemes": ["https"],
  "basePath": "/v1",
  "paths": {
    "/media/{media-id}": {
      "GET": {
        "parameters": [
        {
          "in": "query",
          "name": "access_token",
          "description": "A valid access token.",
          "required": "required"
        }],
        "responses": {
          "200": {
            "description": "Media resource information.",
            "schema": {
              "$ref": "#/definitions/Media"
        }}}
      }
    }
    ...
  },
  "definitions": {
    "Media": {
      "properties": {
        "id:": {
          "type": "string"},
        "users_in_photo": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/Users_in_photo"}
          "link": {
            "type": "string"},
          ...
        },
        "type": "object"
      }
      ...
    }
  }
```

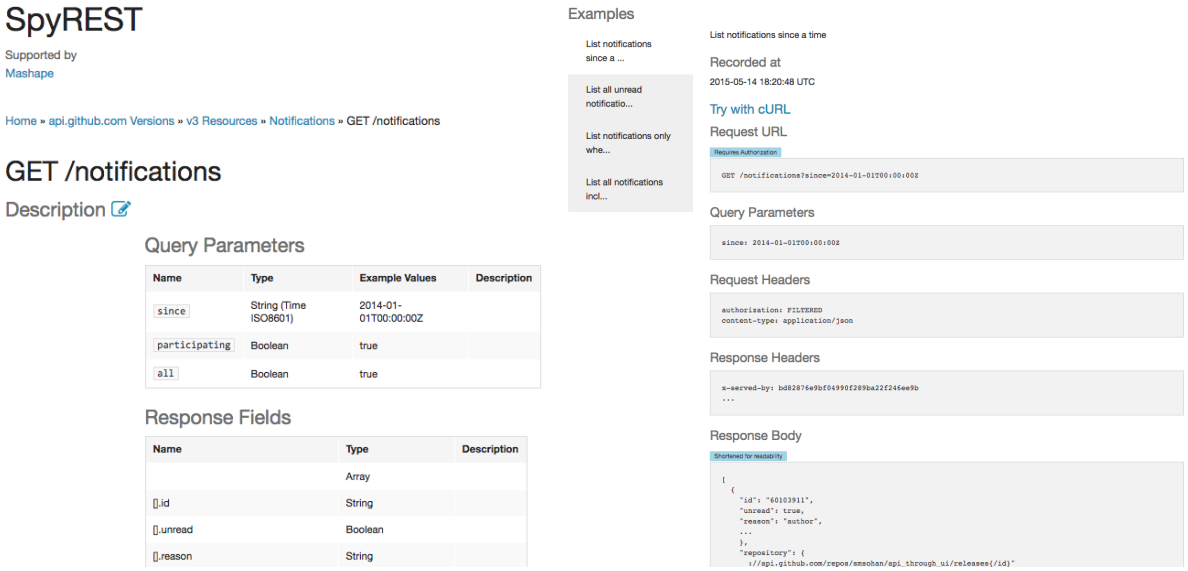
Figure 2.3 – Extract of an OpenAPI specification (generated by our approach) for Instagram

proxy server capture and analyze multiple raw HTTP requests and responses, and then generate the corresponding API documentation. As shown in Figure 2.4, it contains a static description (e.g., query parameters, request headers) showing how to call the endpoint API 2.4a. Also, it provides dynamic examples that allowing developers actually send the requests and retrieve responses 2.4b.

Sohan et al. [Sohan et al., 2017] applied SpyREST to generate and maintain API documentation at Cisco over the eighteen months. Results show that it reduces the massive manual effort when generating the test code. Also, it helps developers to maintain always-updated documentation when the API evolved.

SpyREST is the semi-automated approach. It requires a client that knows how to call the REST services and also requires the client to perform all the possible calls. Also, the actual output of *SpyREST* is the REST service documentation, and thus developers miss the benefit of REST service specification.

SpyREST is dynamic as it listens to the communications that are performed with the REST services to generate the documentation. However, it may not generate the complete documentation due to several restraints, such as the availability of endpoints, authentication, network issues, etc.



(a) API documentation summary (b) API documentation example

Figure 2.4 – A screenshot of SpyREST. It shows a part of auto-generated API documentation and examples for Github service.

APIDiscoverer. Ed-douibi et al. [Ed-douibi et al., 2017] provide *APIDiscoverer*, an example-driven approach for generating REST API specification by using the API call (requests and responses) examples. As shown in Figure 2.5, developers need to fulfill the API calls information and send them to the REST services. It then analyzes requests and responses to generate the specification.

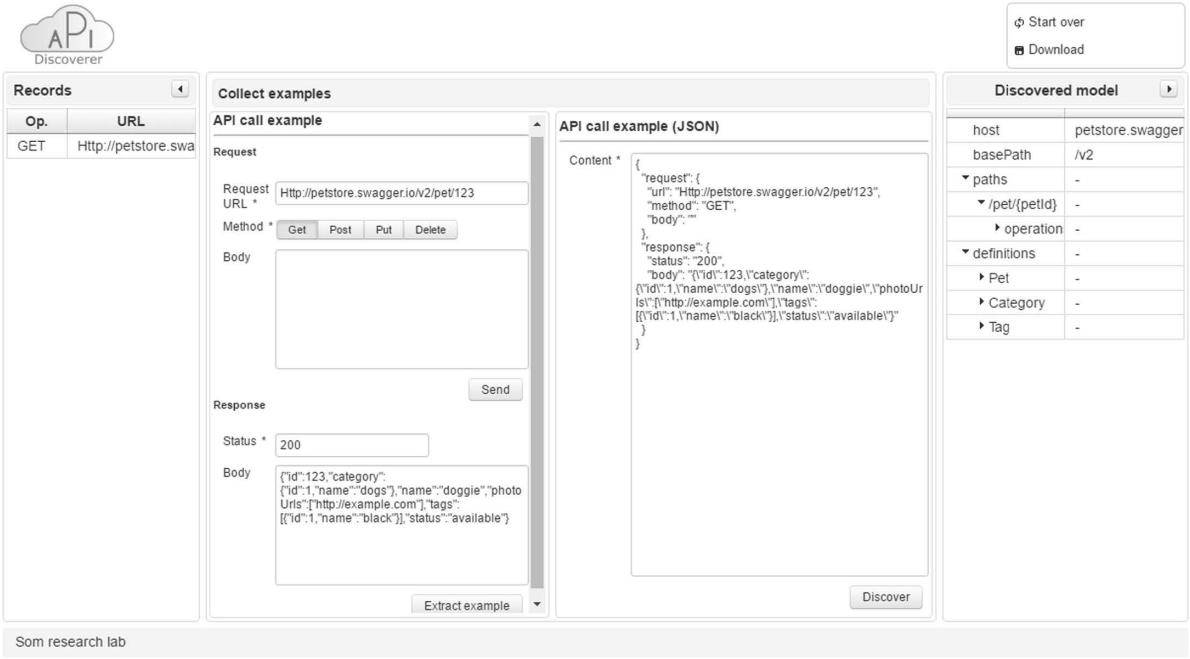


Figure 2.5 – Screenshot for APIDiscover [Ed-douibi et al., 2017].

This idea is quite similar to the SpyREST, but the output description is the OpenAPI specification, which is a benefit for the developers. Another advantage is that APIDiscoverer can generate the JSON schemas of REST resources. It parses the response JSON data, and discovers the JSON schemas that represent the REST resources.

RESTler. Alarcon et al. [Alarcón and Wilde, 2010] provide RESTler that crawls a RESTful service and aims to generate a map that presents all the provided resources and their links. It has proposed a new description ReLL (Resource Linking Language) to describe RESTful services. As a comparison, RESTler visualizes the REST resources while others generate the human-readable documentation or specification. Moreover, RESTler is not an open-source project, and the authors don't present its performance.

D2Spec. Yang et al. [Yang et al., 2018; Dolby et al., 2018] provide D2Spec that extract web API specification from online documentation, which is quite similar to our work. D2Spec

first crawls all the documentation pages of the REST services, and use several machine-learning methods to identify the base URL, path templates and HTTP methods, which are necessary to compose an OpenAPI specification.

D2Spec collects all the URLs in the documentation pages, and use machine-learning techniques to classify the URLs that are relevant to API description. It then uses an agglomerative clustering algorithm to extract the base URL and path templates. D2Spec achieves high precision and recall on extracting the basis specification parts. Also, it can help the API consumer to point out the inconsistencies between online documentation and existing specifications.

However, D2Spec doesn't target the request parameters, which are also essential to the API consumer. Also, D2Spec miss to understand the data returned by the APIs. Web application developers are eager to have the data structures, sample responses to better understand the REST service.

Comparison. We present a comparison with four existing approaches: SpyREST, APIDiscoverer, RESTler, and D2Spec, as shown in Table 2.2. The example-driven approaches SpyREST and APIDiscoverer are quite similar. Developers need to manually search and input API calls information (e.g., URL, HTTP Verb) within corresponding HTML documentation. To get the whole spectrum, developers also need to recursively find the API calls for all endpoints. This work is time-consuming and labor-consuming. It is acceptable for small services that only contain several endpoints (e.g., Instagram includes 21 endpoints). For large services that involve more than 200 endpoints (e.g., Facebook [12] includes 306 endpoints), it becomes annoying and error-prone. The difference between the two approaches is that SpyREST outputs ad-hoc documentation while APIDiscoverer generates a standard OpenAPI specification.

The crawler-based approaches RESTler and D2Spec are feed with an index URL of the online documentation. By automatically crawling the HTML documentation, developers don't need to manually input all API calls. The output parts of these approaches are not complete.

2.2.3 Crowd-sourcing approach

Some REST API specifications are created through the crowd-sourcing approach. Individuals and API providers can submit API specifications to a public available repository. For each REST API, the repository reserves sole and latest specification. The specifications are obtained through the manual effort of API users. Two third-party websites such as API Stack [13] or APIs.guru [14] offer directories of OpenAPI specifications for REST APIs. As an

12. https://developers.facebook.com/docs/graph-api/
13. http://theapistack.com/
14. https://apis.guru/

Table 2.2 – Comparison of existing automated/semi-automated approaches to build the REST API specification.

	Input	Output format	Output parts
SpyREST	API calls	Documentation	Base URL Path Templates Verbs Parameters
APIDiscoverer	API calls	Specification	Base URL Path Templates Verbs Parameters Response Schema
RESTler	Index URL	Structured map	N/A
D2Spec	Index URL	Specification	Base URL Path Templates Verbs

example, Figure 2.6 presents a list of REST services that we can search and directly take advantage of their related OpenAPI specifications.

The advantage of the crowd-sourcing approach is that some specifications in the repository have high quality, compared to the automated approaches. Since they are provided and maintained by REST API providers. Also, developers don't need to take time to generate the specification if it's already in the repository. However, the qualities of the involved OpenAPI specifications vary a lot since they have been provided by different users. Furthermore, it is possible that they are outdated since these directories only rely on API users.

2.3 How to adapt to the data changes of REST services?

Developers want to handle the changing REST services data in a more efficient way. In this section, we first demonstrate the pull mode and push mode to communicate with the REST services. Then we list the existing transformation platforms and their common design mechanism. In those platforms, the fundamental principle is to generate the JSON Patch. In the end, we present the JSON Patch and existing JSON Patch algorithms.

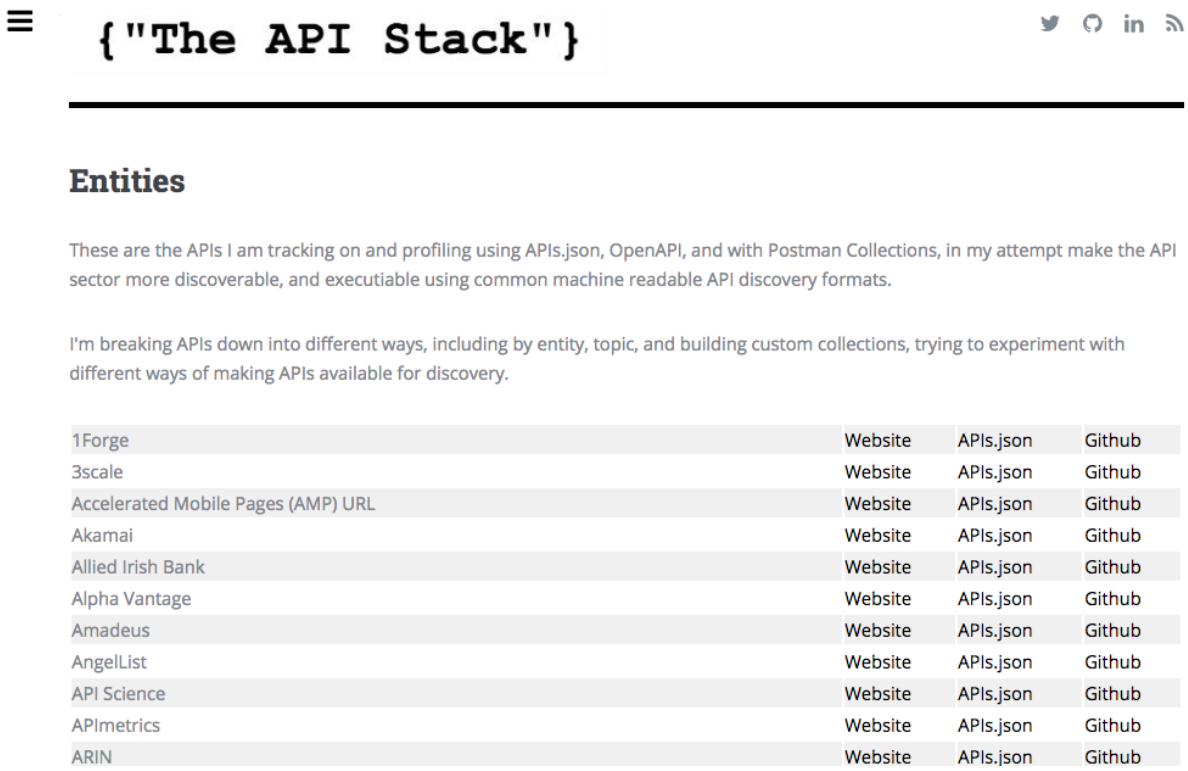


Figure 2.6 – Screenshot for API Stack.

2.3.1 Pull mode and Push mode

Pull mode. The pull mode, or pull technology, is a network communication style that the clients periodically send the requests to the server to get the up-to-date information. REST APIs have been however designed to be used in a pull mode request. For example, Figure 2.7a demonstrates how the client employs pull mode requests to fetch a timeline of tweets from Twitter REST service. In the beginning, the client sends a poll request to the Twitter server, and receives the timeline tweets version 1. The client has actually to call the service according to a predefined frequency even though the timeline tweets have not been changed (see the client sends poll request #2 and receives the same data). Ultimately the client receives the up-to-date tweets version 2 after several requests.

The pull mode is inadequate for services that provide access to data that frequently change. The rates of data change various a lot depending on the services. According to the investigation for RSS blog [Lee, 2012], 11% of bloggers post daily. While Donald Trump tweets on average 15 times a day with a high of 87 [Hersak, 2017]. Setting the pulling period is a trade-off between the promptness of data and the bandwidth cost. The more aggressive the checking updates strategy, the more waste of CPU and network resources, and causes a surge in the traffic [Sia et al., 2007]. Existing literature suggests optimal polling rate for

each service instead of a single common one [Lee, 2012]. However, it's difficult for the web developers to find an optimal polling rate. Moreover, if the developers just want to be aware of new tweets appearing in the timeline, they also have to create the patch describing the differences between the data they previously received and the new one just returned by the request, which can be highly complex depending on the structure of the JSON documents contained in the response of the request.

Push mode. The pull mode, or push technology, is the other network communication style that the client subscribes to the server and they will receive notification messages only when the data have changed. For example, Figure 2.7b presents how the client receives the notifications of timeline tweets in push mode. The client first registers to the Twitter services and receives the timeline tweets version 1. Without endless requests, the client just waits for the notification messages when the tweets have changed. Further the messages contain the set of changes performed to the data rather than the new version of the data, letting the client react to them if needed.

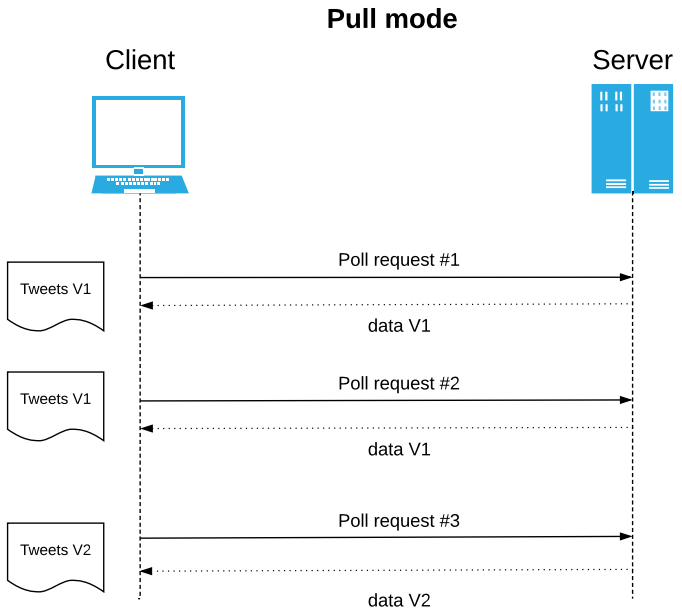
Apple's first introduced a push notification service for the iPhone in 2008 [Melanson, 2008]. It allows developers to push textual messages to iPhone through their services, which can save battery and improve performance. For other mobile platforms, Google announced Google Cloud Messaging [15] and the newest Firebase Cloud Messaging [16] to notify client application of notification messages. W3C also define a standard web push protocol, which allows the server to interact with the client through a push service [Beverloo et al., 2017].

All of the above push technologies are prepared for the REST service providers. They focus on helping REST service providers how to design their services to enable push mode. However, most REST service providers designed their services to be used in a pull mode according to our investigation. As evidence, for the 30 topmost popular REST services, none of them enables push mode requests. The existing circumstances trouble the consumer side web developers since they prefer push mode services, but they don't have access to change the infrastructure of those services. Hence, web developers are eager to have a platform that transforms a pull mode service to a push mode one without modifying the source codes.

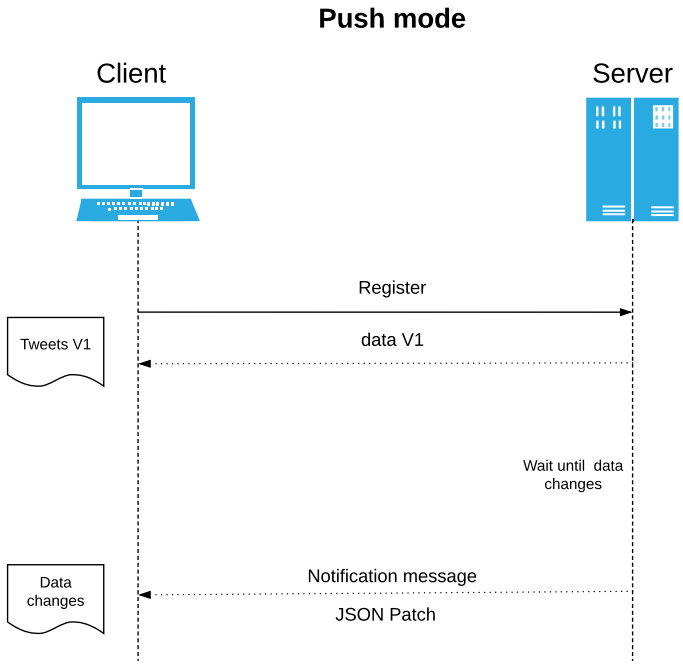
2.3.2 Transformation platforms

Some commercial companies already found and support the need for transforming the pull mode service into a push one. We take a brief overview of these transformation platforms.

15. https://developers.google.com/cloud-messaging/
16. https://firebase.google.com/docs/cloud-messaging/



(a) Pull mode request



(b) Push mode request

Figure 2.7 – Two modes for requesting the Twitter REST services

StreamData.io. Launched in 2015, our partner StreamData.io [17] is a French-based company that provides a proxy server to convert the pull mode API of an existing web application into push mode one. Once the client registers on the platform, the proxy server will take the place of the client to poll the destination REST API, and analyze the response data (normally in JSON format). The proxy server then compared the stored previous version and the newest version of data, and generate a corresponding JSON patch for the two JSON documents. When nothing has changed, the platform didn't transfer useless data to the client. When the data has updated, the platform only sends back the JSON patch, which is smaller than the newest data in size. This approach doesn't save bandwidth cost and system resource cost for the proxy server since it needs to polling the REST services continuously. However, for the client part, the load and bandwidth can be reduced by up to x66 according to StreamData.io [streamdata.io, 2016].

Figure 2.8 – A screenshot of Streamdata architecture. It acts as a proxy server to polling the REST API and returns JSON Patch to the client only when source data has been changed. The client then can apply the JSON Patch to recover the up-to-date data.

The most difficult part of this platform is to create a patch between each successive received versions of the data. Existing solutions have some imperfections on generating the JSON patch. Section 2.3.3 discusses this issue in detail. We provide a new algorithm for StreamData.io that can generate the JSON patch in a smaller size and within less time. Our contribution would improve the performance of this transformation platform.

Diffusion. Founded in 2006, Diffusion [18] is an intelligent platform that provides real-time data streaming via the push mode. Similar to StreamData.io, it also acts as a proxy server to manage and synchronize data. It also supports two-way communication that enables the client to publish a topic and synchronize it with the server. Diffusion is not solely designed

17. http://streamdata.io/
18. https://www.pushtechnology.com/

for converting REST services, since its target data can be stemmed from any sources (e.g., REST services, database). Developers may need additional configurations to fetch the data from REST services.

Even though Diffusion claims that it greatly reduces the bandwidth requirements and latency, and offers up to 90% data efficiency improvement [pushtechnology, 2018]. We made an investigation toward its JSON patch algorithm and found that it only supports the simple add, remove, and replace patch operations.

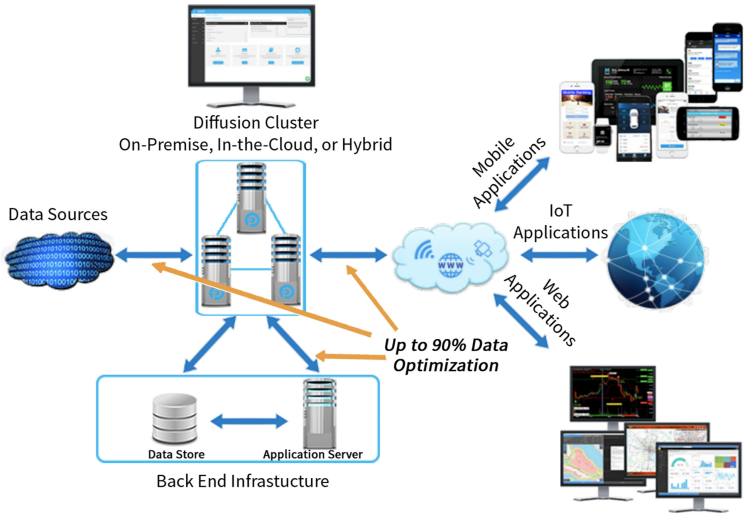


Figure 2.9 – A screenshoot of Diffusion architecture. It provides a cluster that can provide optimal JSON data to endpoint users (i,e., Mobile application, Web application, IoT application.).

As a summary, existing transformation platforms follow the same principle that acts as a proxy server to transforming the polling requests into the push notifications. The main difference and the most difficult part is the JSON Patch algorithms they used. In the following section, we will discuss the JSON document and JSON Patch in detail, and compare the existing JSON Patch algorithms.

2.3.3 JSON document and JSON patch

In this section, we give a description of the JSON document and compare the existing JSON Patch algorithms.

JSON document. A JSON document is a very simple textual serialization of a JavaScript object. More precisely, it is a tree composed of three kinds of nodes (literal, array or object), where the root node cannot be a literal. A literal node can be either a boolean, a number or

a string. An array node is a sequence of nodes. An object node has a set of child properties, each of them has a label [19] unique within the object, and a value that is a node. As an example, the Figure 2.10 presents two simple JSON documents that contain literals, objects and arrays.

```
{
  "isOk": true,
  "rm": "2",
  "val": 3,
  "mes1": {"who":"me", "exp":0},
  "res": [
      "v1",
      "v2",
      "v3",
      "v4",
      "v5"
  ],
  "inner" : {
    "elts" : ["a","b"] ,
    "sum" : "test is ok"
  }
}
```

```
{
  "rank": 6,
  "isOk": false,
  "va": 3,
  "mes1": {"who":"me", "exp":0},
  "mes2": {"who":"me", "exp":0},
  "res": [
      "v6",
      "v1",
      "m2",
      "v1",
      "v5",
      "v3"
  ],
  "inner" : {
    "in": {
      "elts" : ["a","b","c"]
    }
  },
  "sum" : "test is ok"
}
```

Figure 2.10 – A *source* (left) JSON document with several properties. A *target* (right) JSON document that has been transformed from the *source* JSON document.

JSON Patch. The JSON Patch RFC is an ongoing standard that specifies how to encode a patch that can be performed on a JSON document to transform it into a new one[Bryan and Nottingham, 2013]. The RFC specifies that a patch is a sequence of change operations. It then specifies the five following change operations (a sixth operation is defined to perform tests):

— Add: this operation is performed to add a new node into the JSON document. The new node can be added within an array or as a new property of an object.

— Remove: this operation is performed to remove an existing node of the JSON document.

— Replace: this operation is performed to replace an existing node by another one.

— Move: this operation is performed to move an existing node elsewhere in the JSON document.

19. A string or a JavaScript name.

```
[
  { "op": "add",     "path": "/rank",  "value": 6 },
  { "op": "remove",  "path": "/rm"},
  { "op": "replace", "path": "/isOk",  "value": false},
  { "op": "move",    "path": "/va",    "from": "/val"},
  { "op": "copy",    "path": "/mes2",  "from": "/mes1"}
  { "op": "add",     "path": "/res/0", "value": "v6"},
  { "op": "replace", "path": "/res/2", "value": "m2"},
  { "op": "remove",  "path": "/res/4"},
  { "op": "copy",    "path": "/res/3", "from": "/result/1"},
  { "op": "move",    "path": "/res/5", "from": "/result/4"},
  { "op": "move",    "path": "/inner/in/elts",   "from": "/inner/elts"},
  { "op": "add",     "path": "/inner/in/elts/2", "value": "c"},
  { "op": "move",    "path": "/sum",   "from": "/inner/sum"}
]
```

Figure 2.11 – A RFC JSON Patch that, if applied to *source* JSON document of the Figure 2.10, would get the *target* JSON document.

— Copy: this operation is performed to copy an existing node elsewhere in the JSON document.

The RFC specifies a standard way to encode a patch into a JSON document. More precisely a patch is an array of change operations where each change operation is encoded by a single object with properties specifying the kind of operation, the source and target nodes, and the new value if needed. For instance, the Figure 2.11 presents a patch that can be applied to *source* JSON document presented in the Figure 2.10, and that contains change operations (adding a new literal node $rank$, removing a node of the array res, etc.). We use that example in the following sections.

Applying a patch to a JSON document is quite easy. It consists in applying all the editing operations of the patch in their defined order. Creating a patch that, given two versions of a JSON document, expresses how to transform the first version into the second one is however much more complex, especially when the goal is to create small patches and to create them as fast as possible.

2.3.4 JSON Patch Algorithms

JSON documents are mainly labeled unordered trees (object nodes and their properties), where some nodes are arrays, hence ordered. The theory states that when just the add, $remove$ and $replace$ operations are considered, the problem of finding a minimal patch is $O(n^3)$ for ordered trees and NP-hard for unordered trees [Zhang et al., 1992; Bille, 2005; Pawlik and Augsten, 2011; Higuchi et al., 2012]. When the $move$ operation is also considered, the problem is NP-hard for both kind of trees [Bille, 2005]. That is why several algorithms from the document engineering research field use practical heuristics. One of the most famous is the algorithm of Chawathe et al. [Chawathe et al., 1996] that computes

Table 2.3 – Comparison of existing approaches to generate the JSON Patch.

		Libraries	
Category	**Scenario**	*move* and *copy*	*Arraynode*
JavaScript	`jiff`	No	Stack
	`Fast-JSON-Patch`	No	No
	`JSON8 Patch`	No	No
	`rfc6902`	No	Stack
Python	`python-json-patch`	No	Stack
PHP	`json-patch-php`	No	No
Java	`json-patch`	Yes	No

patches (containing move actions) on trees representing LaTeX files. Several algorithms have also been designed specifically for XML documents [Cobena et al., 2002; Al-Ekram et al., 2005]. One of them [Lindholm et al., 2006] is even capable of detecting copy operations.

Several existing approaches support the creation of JSON Patches. [20] By analyzing all of them, it appears that they all take one or two of these simplifications to make the problem tractable (see Table 2.3):

— They choose not to support the *move* and *copy* operations that are yet specified in the RFC, and therefore provide non-optimal patches. As an example in the Figure 2.11, an optimal patch uses *move* operation to handle the property label renaming from *val* to *va*. Without such a *move* operation, the patch then uses a *remove* property *val* and a *add* property *va*. Moreover, an optimal patch uses a *copy* operation for the property *mes*2 and its value copied from *mes*1. The Table 2.3 shows that only one existing approach does support these operations.

— They choose not to support array node, or to support them poorly. In principle all the editing operations of the JSON RFC apply to array nodes as well as object nodes. A patch can then express changes done within an array. For instance in Figure 2.11, an optimal patch uses the *move* operation to put $v3$ to the end of the array. Moreover, it uses the *copy* operation for copying the existing node $v1$. Regarding the support of array, the Table 2.3 shows that half of the approaches do not support array at all, and consider them as a simple node (with nothing inside). The other half simply considers that an array is a stack, and therefore supports change operation that can apply to a stack (*push* and *pop*).

20. http://jsonpatch.com/

The table 2.3 clearly shows that there is no approach that fully complies with the RFC in terms of change operation coverage. By compliance we mean that it can handle all editing operations that are defined by the RFC including the move and copy ones (the test one is not an editing operation). However there is no formal process that truly checks the RFC compliance. There is only JSON test [21] that just checks if the given patches can be applied.

2.4 Summary

The state of the art brings a lot of information about the usage of third-party components in the context of web applications. As a summary, we list the three main lessons that are learned from the existing literature:

— Our first problem, which is about JavaScript library identification and recommendation. we summary the existing literature into two aspects: obtain third-party library usage dataset, and sort the libraries according to underlying requirements. For the first step, existing studies rely on open source projects while we want to target famous web applications, which are often closed-source applications. For the second step, existing approaches focus on the scenario where developers want to find similar libraries to replace the used ones. We want to concentrate on the scenario where developers want to choose appropriate libraries when they design the web applications. We choose to use library popularity among famous web applications to rank the libraries. The underlying assumption is that the "best" libraries should be the ones that used by the majority of people.

— Our second problem, namely the generation of standard REST specification, has been studied by several researchers. Existing approaches relay a lot of manual human efforts to generate and maintain the specification. Moreover, they fail to create some parts of the specification that are crucial to the web developers. This phenomenon motivates our second contribution that can reduce human efforts and generate a complete specification at the same time.

— Our third problem, related to the adaption to the changing data of REST service, has aroused the interest of the industry. They provide a solution that can transform from a classical pull mode request to a modern push one. Even though they follow the same principle for conversion, the fundamental algorithm they use is not perfect. The existing JSON Patch algorithms do not generate optimal patches, and not take advantages of some patch operations that defined in the JSON Patch RFC. We want to address this problem via a new efficient patch algorithm that fully complies with JSON Patch RFC.

21. https://github.com/json-patch/json-patch-tests

CHAPTER

3 What are the best JavaScript libraries to use?

Modern web applications often use JavaScript libraries, such as jQuey or Google Analytics for example, that make the development easier, cheaper and with better quality. Choosing the right library to use is however very difficult as there are many competing libraries with many different versions. To help developers in this difficult choice, popularity indicators that pinpoint which applications use which libraries are very useful. Building such indicators is however challenging as popular web applications usually don't make their source code available. In this chapter, we address this challenge with an approach that automatically browses web applications to retrieve the JavaScript libraries they use. By applying this approach to the most famous websites, we then present the trends we observed and the recommendations.

Contents

3.1 Introduction

In this chapter, we discuss the problem introduced in Section 1.2.1. As a reminder, it refers to the following scenario: a developer wants to choose which JavaScript library to include in the web application.

Software projects often use third-party libraries. With Java projects for example, 70% of the projects use at least four third-party libraries, and 10% use more than ten libraries [Teyton et al., 2014]. For web applications, the growing interest for library management systems, such as NPM with NodeJS, shows the importance of this topic[Mardan, 2014]. Further, it is well known that third-party libraries such as *jQuery* are almost used by all web applications [Bibeault and Katz, 2008].

To help developers to chose the right library to include, many existing research approaches aim to provide popularity indicators on libraries [Bauer et al., 2012; Kula et al., 2015; Teyton et al., 2012, 2013; Thung et al., 2013; Zimmermann et al., 2005]. In order to get the popularity indicator, we need to recognize the JavaScript libraries used in web application. However, currently there is no publicly available approach since popular web applications usually don't make their source code available.

In this chapter, we overcome this difficulty and exhibit which are the popular JavaScript libraries. Our proposal performs an online observation of popular web applications with the underlying assumption that the libraries they use are most probably the ones that should be used. The main difficulty is therefore to identify which are the libraries they use just by browsing them. To get significant results we decide to observe the Alexa global top 100 websites [1]. Our approach then browses these web applications and recognizes the JavaScript libraries they use. Based on such observations we can then output trends and give recommendations.

We make the following contributions:

— We provide an automatic and efficient approach that browses web applications and detects their used JavaScript libraries (see Section 3.2). Our approach uses both syntactical and dynamical analysis of the online resources of the web applications and returns high precision results (see Section 3.3 and Section 3.4).

— By applying our approach on the 100 most popular websites, we provide statistics of the use of JavaScript libraries. Such statistics confirm the fact that libraries are largely used in web applications (see Section 3.5).

— We then present how our observations can be used to provide trends and recommendations. As an illustration, we present our observation by focusing on *jQuery* and *AngularJS*, *Backbone* and related libraries (Section 3.5).

1. http://www.alexa.com/topsites

Table 3.1 – Library usage matrix.

	JavaScript Libraries	
Web applications	jQuery	AngularJS
Alipay	$\perp$	?
Dropbox	1.3.1	$\perp$
Pixnet	$\perp$	$\perp$

3.2 Methodology

This section first provides definitions for web applications and libraries. It then presents three strategies that can be used to detect the libraries used by a web application. Then it presents our global approach that uses these three strategies.

3.2.1 Definitions

First, we consider that a web application is a pair (w, url) where w is the name of the web application and *url* is its root url. For instance, (DropBox, www.dropbox.com) is a web application. Second, we consider that a JavaScript (JS) library is a pair (l, V_l) where l is the name of the library and V_l its corresponding ordered set of versions. For instance (JQuery, $\{1.1, 1.2, 1.3\}$) is a library.

As we briefly presented in Section 3.1, our objective is to automatically identify the JS libraries used by web applications. Table 3.1 shows a tiny example of the result we want to obtain, called the *library usage matrix*. The library usage matrix contains a set of web application names as rows, and a set of JS library names as columns. The goal of this table is to indicate which library is used by which web application, and to give the corresponding version. Therefore the value of a cell corresponding to an application w and a library l belongs to the set $V_l \cup \{?, \perp\}$, where V_l is the set of all versions of library l, as previously defined, and where the symbol ? means that w is using an unknown version of l, and where $\perp$ means that the application is not using the library. Table 3.1 states for instance that *DropBox* uses only the version 1.3.1 of *jQuery*, *Alipay* uses an unknown version of *AngularJS* and *Pixnet* does not use any library. We will use the library usage matrix to compute which libraries are popular, and which particular versions of the libraries are popular, with the main objective to exhibit trends and to give some recommendations.

3.2.2 Recognition Strategies

Filling the library usage matrix requires to browse web applications and to recognize which libraries they use. It is done by using several *recognition strategies* that are programs

that take as an input a given web application, and produce as an output a row of the library usage matrix. The remainder of this section presents the three different recognition strategies we propose that, when combined, provide good results (see Section 3.4).

Comment Strategy

The idea behind the *Comment Strategy* is to search for names of libraries in the header comment of the JS files used by a web application. This strategy is quite efficient since library files often contain this information. For instance, Figure 3.1 shows the three first lines of a Modernizr JS library file. We clearly see in this header that this file cites the name of the library (Modernizr) and its version (2.8.3).

To fetch this information, the comment strategy begins by browsing the root URL of the web application. Browsing this URL returns all the webpage content, including a set of linked resources. We retain from this set of resources only the ones that correspond to JS files, by using the *content-type* information. We then use several regular expressions to extract the library name and version from the comments, as shown in Figure 3.2. More precisely, one regular expression is generated for each sought library by substituting the variable $name$ by the name of the sought library. These regular expressions are then all checked, in global mode, against a JS file until one matches, ignoring the case. This regular expression is robust enough to match comments such as *jQuery JavaScript Library v1.11.3* or *Modernizr v2.8.3*.

The comment strategy has the main advantage to be very efficient since executing a regular expression is quite fast. Furthermore, this strategy can be easily extended to support the search of new libraries just by adding new library names. Unfortunately, several web applications remove all the origin comments of a library file, which somehow leads to no results by using this strategy. The detailed performance results are presented in the Section 3.4.

```
/*!
* Modernizr v2.8.3
* www.modernizr.com
```

Figure 3.1 – The header of a Modernizr JS library file.

```
name\s(.*\s)?v?([0-9]+)(\.[0-9]+)*
```

Figure 3.2 – The regular expression used to extract library names and version from comments.

File Matching Strategy

The intuition behind the *File Matching Strategy* is to check if a JS file used by the web application is similar to a file that is known to be a JS library. Similarly to the first strategy, this strategy starts by retrieving all the JS files used by the web application. It also uses a so-called *knowledge base* which contains the files of all versions of all JS libraries. This knowledge base is large: it contains more than 2000 files in our current implementation.

Asking for an extensive comparison with all the files of the knowledge base would take too long. To perform such a comparison in a reasonable time, we use a two-step processIn the first step, we use simhashes, which are short hashcodes computed on a large text, and that can be used for quick comparisons [Sadowski and Levin, 2007]: the more similar the two texts, the more similar their simhash. Therefore, prior to any comparison, we compute a simhash for all the files of the knowledge base. When analyzing a JS file that is used by a given web application, we then generate its simhash and compare it, using the Hamming distance [Hamming, 1950] to all simhashes of the knowledge base. We retain only the files from the knowledge base that have a Hamming distance $d \leq 3$. The thresholds of our strategy will be discussed in Section 3.4.

Since the multiple versions of a same library are usually very close, this process generally retains several files from the knowledge base. To retain only one result, we perform a more detailed comparison in the second step. We then compute the Dice coefficient for each candidate file on the bigrams they contain[Dice, 1945]. We retain only the files having a Dice coefficient $c \geq 0.8$, and among these files we retain only the file with the greatest coefficient. If there are several files with a same maximum coefficient, we distinguish two cases: 1) if the files do not come from the same library, we return no library and 2) if the files come from the same library, we return the file associated to the greatest version.

The file matching strategy has the main advantage to be robust to small modifications of the library, as it is sometimes done by web developers. It is quite easy to extend as it only requires the set of library files. Its first drawback is its cost in time. However, thanks to our optimization, the total comparison cost time is reasonable as we will measure it in Section 3.4. Its second drawback is that some web applications modify the source code of the library or merge several libraries into one JS file. In this situation this strategy fails.

Sensor Strategy

The *Sensor Strategy* aims at inserting at runtime a sensor in the web application with the objective to dynamically detect which JS libraries are deployed. Such a sensor is a JS plug-in that is executed by the browser. To detect a library, the sensor uses two elements. First it monitors the requests made by the browser in order to detect URLs (called key URLs) associated to a known library. Then it checks for the existence of JS objects (called key objects) that are associated to a known library.

For instance, if the sensor detects that the web application is requesting the *con-*

Table 3.2 – Key URLs and objects for several libraries.

	Element	
JavaScript library	Key URLs	Key objects
jQuery	jquery.com	*window.jQuery*, *window.$*, *window.$jq*
Modernizr		*window.Modernizr*
Facebook SDK	connect.facebook.net	*window.FB*
Twitter Platform	platform.twitter.com	*window.twttr*

nect.facebook.net URL, it means that it is using the *Facebook SDK* library. Similarly, if the sensor detects that a *window.$* object exists, it means that the application is using *jQuery*. Table 3.2 shows key URLs and objects for several well known libraries.

When a library is detected thanks to a key URL or object, our sensor then calls a specific function that aims to detect the version of the library. Such a function uses the internal knowledge of the library to recognize the version, i.e. objects specific to a particular version or functions that return the version. Further, the function returns ? when no version can be detected.

The sensor strategy has the advantage to be quite efficient. Its main drawback is that it requires a lot of configuration that has to be provided by an expert that knows the internal of a library. Therefore adding a new library in the knowledge base is expensive. Moreover, sometimes the libraries do not provide a mean to distinguish between its different versions (for instance same URLs are requested, and the same objects are defined whatever the versions). Further, as it relies on executing JS code coming from the web applications and the plugins, it sometimes fails due to some unexpected runtime error.

3.2.3 ARJL Combined Strategy

The three strategies we presented above give different results according to the input web application and to the sought library. For instance, Table 3.3 shows the results obtained by these three strategies on several couple of applications and libraries. In this table, we see that the library *jQuery* is detected with the same version by the three strategies for the Microsoft website. However, in the 360 website, *jQuery* is detected only by the two first strategies. Also in the Microsoft website, *Sizzle* is not detected by the sensor strategy, and detected in two different versions by the comment and file matching strategies.

Section 3.4 gives much more information regarding the precision of each of the strategy. However, we clearly see that there is no silver bullet: no strategy always return a correct result.

Table 3.3 – Comparison of three recognition strategies.

	Strategy		
Web application / JavaScript library	Comment	File Matching	Sensor
`Microsoft / jQuery`	1.7.2	1.7.2	1.7.2
`360 / jQuery`	$\perp$	1.7.1	1.7.1
`DropBox / Modernizr`	2.8.3	$\perp$	2.8.3
`DropBox / Underscore.js`	1.8.3	1.8.3	$\perp$
`Microsoft / Sizzle`	1.9.4	1.9.2	$\perp$

We can see that the results obtained by the three strategies should be merged. We therefore introduce ARJL (Automatic Recognizer of JS Libraries) that integrates the three strategies mentioned above, as follows. First the comment, file matching and sensor strategies are applied separately on a given web application. For each library l, we therefore get three results in the set $V_l \cup \{?, \perp\}$. Recall that V_l is a totally ordered set, i.e. $1.0 < 1.1 < 1.2$. We extend the total order of V_l by considering $\perp$ as the smallest element, and ? the second smallest, i.e. $\perp <? < 1.0 < 1.1 < 1.1$. Using this total order, for a library and a given web application, the ARJL strategy returns the greatest element in $V_l \cup \{?, \perp\}$ that has been computed by the strategies. For instance, in Table 3.3, for the *Microsoft* application and *Sizzle* strategy, we have $\perp < 1.9.2 < 1.9.4$ therefore 1.9.4 is returned.

3.3 Implementation

Our approach has been implemented using the JavaScript (JS) language on top of the SlimmerJS [2] scriptable and headless (without GUI) web browser. SlimmerJS is in charge of applying the three strategies. In particular, we use it to retrieve the JS files from the web applications (for the comments and file matching strategies), and also to look for the key URLs and objects (for the sensor strategy). Then, we apply the ARJL strategy on top of the obtained results.

Since SlimmerJS can experience a heavy network traffic when crawling a web application, or even crash when executing the remote JS code, we analyze each web application in a separate thread. A monitor watches all the threads and when one is running for too many time (we have a configurable threshold set by default to five minutes), it kills and relaunches it. Fortunately, we always succeed in analyzing all the web applications we wanted to analyze as the crashes rarely happen.

Since there exists a huge amount of JS libraries to detect, we had to choose a reasonable subset of them to test our approach. We therefore chose to select a limited set of famous

2. https://slimerjs.org/

JS libraries from two well-known sources. First, we chose to include the 14 JS libraries stored by the Google CDN [3]. Second, as Wikipedia [4] provides a list of notable JS libraries, we choose this list but remove the libraries without any update in the last five years or the ones that are totally abandoned. By merging these two lists, we obtain a list of 52 JS libraries, with an average of 43 versions per library.

To build the knowledge base of all files of all versions of these libraries, we use the CDNJS [5] library hosting website. To elaborate the key objects and URLs for the sensor strategy, we reused the source code of Library Detector [6], a Chromium plugin that detects the libraries used by a web application. We then have extended it to handle our sensor strategy.

For the list of web applications, we use the Alexa website that ranks websites according to their popularity. Alexa contains websites that are available on several domains (such as *google.com*, *google.co.in*, *google.co.jp*). We therefore remove the domain information from the URL, and select the 100 most popular ones.

To allow researchers and developers to replicate our results, the source code of our approach is available on GitHub [7].

3.4 Evaluation

In this section we evaluate the strategies described in Section 3.2. Firstly, we describe how we set up the two thresholds of the file matching strategy. Then, we analyze the precision of our approach. We also compare the results of the different strategies. Finally, we measure the time performances of the strategies.

3.4.1 Thresholds of the File Matching Strategy

As described in the previous section, the file matching strategy uses two thresholds: the maximum Hamming distance d between simhashes, and the minimum Dice similarity s between the text of the files. To compute these thresholds our objectives was that the strategy should avoid at all costs false positives, and should return the largest set of identified libraries. In other words, the precision should be 100%, and the recall should be as big as possible (close to 1).

We then used a manual process that started with the stronger threshold (where $d = 0$ and $s = 1$), and aimed to release the thresholds to get more identified libraries without having any false positive. In other words, we tried to get the two thresholds with a precision of 100% and with the biggest recall. We ran that process on a set of 10 applications randomly

3. https://developers.google.com/speed/libraries
4. https://en.wikipedia.org/wiki/List_of_JavaScript_libraries
5. https://cdnjs.com/
6. https://github.com/johnmichel/Library-Detector-for-Chrome
7. https://github.com/kenmick/WebCrawler

Table 3.4 – Hamming distance and Dice similarity thresholds, with the associated true and false positives.

	Dice coefficient				
Hamming distance	1	0.9	**0.8**	0.7	0.6
0	(4,0)	(9,0)	(9,0)	(9,0)	(9,0)
1	(4,0)	(10,0)	(10,0)	(10,0)	(10,0)
2	(4,0)	(10,0)	(10,0)	(10,1)	(10,2)
3	(4,0)	(10,0)	**(12,0)**	(12,2)	(12,4)
4	(4,0)	(10,0)	(12,1)	(12,5)	(12,7)

chosen from the 100 most popular application from Alexa. Further one author manually inspected the detected libraries and classified each of them as either a true or a false positive. The Table 3.4 shows the results of our process where the candidate values for d are $\{0,1,2,3,4\}$, and $\{1,0.9,0.8,0.7,0.6\}$ for s. For each couple of thresholds (c,s) in the Table, there is an associated couple (t,f), where t (resp. f) is the number of true (resp. false) positives. According to our considerations, we therefore selected the thresholds $d = 3$ and $s = 0.8$ as it returned the more libraries (12) without any false positive (0).

3.4.2 Precision of the Strategies

We consider two kinds of precision, called the *library-level* and the *version-level* precisions. The *library-level* precision focuses on library name, and ignores the versions. In such a level, a true positive is when a strategy returns a library that is truly used by the web application, whatever the version returned by the strategy and the one that is truly used by the application. A false positive is when a strategy returns a library that is not used by the web application. The *version-level* precision focuses on versions. In such a level, a true positive is when a strategy returns the version of a library that is truly used by the web application. A false positive is when a strategy returns a version that is not used by the web application. When the strategy does not return the version (unknown version), we don't consider that as a result, so it is not a true nor a false positive.

To evaluate these two precisions, we drew at random 20 web applications from the top 90 applications of Alexa (we excluded the ones used in the threshold experiment). We then ran our strategies on these applications and collected the results. One of the authors then checked if the results were true or false positives. To perform this check, the author just used all the development tools of Mozilla Firefox with the intent to check whether the return library is really used by the web application.

Table 3.5 shows the precision of our strategies. Regarding the *library-level*, both the comment and the file matching strategies have a 100% precision. The sensor strategy has a

93.5% precision mainly because few web applications use objects that have the same name than key objects. Regarding the *version-level*, the precision of all the strategies is very close to 90%. All together, the precision of our approach is 94.4% for the *library-level*, and 92.6% for the *version-level*, which is quite good.

3.4.3 Comparison of the Strategies

In this section, we perform a comparison of the results obtained using the different strategies. For this purpose, we ran the comment, file matching and sensor strategies on the 100 most popular applications of Alexa. In this experiment, we removed the version information retrieved by the strategies, so if a strategy returns $\{(jQuery, 1.2)\}$, it is transformed as $\{(jQuery, ?)\}$. Using this transformed data, we construct the Venn diagram of Figure 3.3 that shows the intersections of the results of the strategies. This diagram clearly shows that each strategy is useful because it identifies libraries that are not detected by the other strategies. The sensor strategy seems to outperform the comment strategy that in turns seems to outperform the file matching strategy as they find respectively 78%, 40% and 22% of all libraries. The set of uniquely detected libraries represent 53% of all libraries for the sensor strategy, 13% for the comment strategy, and 8% for the file matching strategy.

Finally Figure 3.4 shows the comparison of '?' and accurate version for each strategy. It shows the number of accurate and unknown versions returned by a strategy. This figure confirms that the comment and file matching strategies never return unknown version, as expected. Only the sensor strategy returns unknown versions. This Figure also shows that the sensor strategy return more versions than the other ones, which also explain the Venn diagram of the Figure 3.3.

3.4.4 Efficiency

In this section, we assess the time performances of our strategies. First, SlimmerJS require 3 hours to browse all the web applications from top 100 applications of Alexa. Then, we measure the total time taken by each strategy to analyze all these applications. This time has been measured using a Intel Core i7-4770 CPU @3.40GHz×8, 16GB of RAM, and Ubuntu 14.04.2 LTS x86 64. We did not measure the time taken to compute the ARJL strat-

Table 3.5 – Precision of the strategies.

	Strategies			
Precision	Comment	File Matching	Sensor	ARJL
Library-level	100%	100%	93.5%	**94.4%**
Version-level	88.9%	94.4%	90.9%	**92.6%**

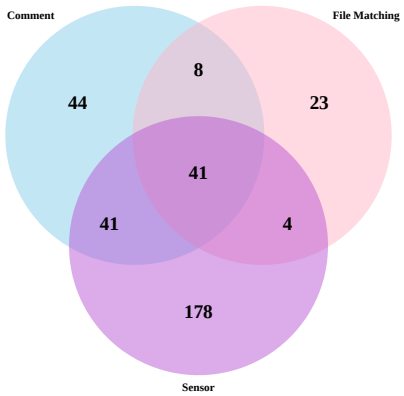


Figure 3.3 – Venn diagram for 3 strategies

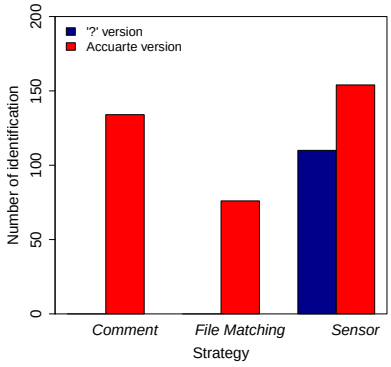


Figure 3.4 – Comparison of '?' and accurate version for each strategies

egy because it only combines the results of the others in a few milliseconds. The fastest strategy is the comment one (0.1 hours). The sensor strategy is the longest one, since it takes two hours and a half to process all sources (2.5 hours). This is because runtime errors that can be experienced when running JS code. Finally, the file matching strategy takes 1 hour. In conclusion, the top 100 websites can be processed in less than 7 hours.

3.5 Observations and Suggestions

This section presents our observations and provides some suggestions regarding the use of JavaScript (JS) libraries in the context of web development. We first present some statistics on how famous web applications are using libraries. Then, we present the suggestions that can be provided looking at a recent snapshot of the library usage of the most famous web applications. Finally, we present our observations on the library usage during

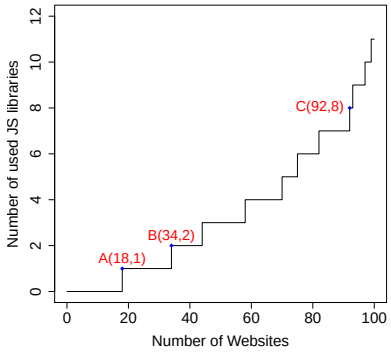


Figure 3.5 – Statistics for Top 100 web applications on Oct. 20, 2015

a long period of time. Such observations yield interesting insights regarding the pace of library evolution.

3.5.1 Statistics

Figure 3.5 presents the global usage of JS libraries by the Alexa top 100 web applications. It shows that most of these web applications use several JS libraries: 82% of them use at least 1 library, 66% use at least 2 libraries, and 8% use at least 8 libraries (see the points A, B and C).

By looking into these web applications, we observed that top ranked web applications use few libraries, even if their company develops and maintains several famous ones. For example, the *Google* web application ranks first but does not use any library, even if the Google company provides widely used libraries such as *GoogleAnalytics* or *GoogleAPI*. On the contrary, web applications that have a lower ranking use more libraries. We have validated this observation by using a Spearman correlation test. The results of this test is $\rho = 0.37$ with a p-value of 0.0006. There is therefore a medium correlation between the rank of a web application in the top 100 and the number of library it uses.

These statistics reinforce our hypothesis that the libraries used by the famous web applications are the ones to use. Indeed as famous web applications use only a few libraries, we claim that they do use the libraries that provide a very strong added-value. However, this phenomenon questions the number of popular web applications that have to be considered to perform valuable observations that yield useful suggestions. We currently choose 100 web applications but we are not sure that this number is representative enough. Therefore this number could be increased or decreased depending on the objective: analyzing only widely used web applications, or analyzing also less famous web applications. We investigated how the results change when using 1000 applications, and there was a very limited impact on the results.

Table 3.6 – JS library usage frequency for Alexa global top 100 web applications on Oct. 20, 2015

Snapshot-2015-10-20	Library name	Frequency
1	jQuery	63
2	GoogleAnalytics	25
3	Modernizr	19
4	jQueryCookie	18
5	Underscore	12
6	jQueryUI	12
7	Facebook SDK	11
8	RequireJS	9
9	SWFObject	8
10	Twitter	7
...		
	Backbone	7
	Bootstrap	3
	AngularJS	0
	Polymer	0

3.5.2 Analysis of the October 2015 Snapshot

Table 3.6 lists the JS library usage frequency for Alexa global top 100 web applications on Oct. 20, 2015. We can observe that *jQuery* is widely used by web applications and ranks first (63 web applications use it). In the opposite MVC (Model View Controler) libraries such as *AngularsJS*, *Backbone* and *Polymer*, are rarely used by famous web applications. These results contradicts the trends provided by JS.ORG [8], which gives statistics about popular JS projects on GitHub. Table 3.7, which presents the statistics of JS.ORG, states that *AngularsJS* ranks first which is very different from the results of our study.

We claim that our results confirm our hypothesis, and that they can be used to give some suggestions. In particular, we claim that libraries such as *jQuery* are essentials. On the contrary, libraries such as *AngularJS*, *Backbone* and *Polymer* should be avoided as they are possibly not yet mature enough to be included in web applications.

Our observations that focus on versions can be used to give precise suggestions on a particular library. As an example, Figure 3.6 shows the version distribution of the *jQuery* library among the top 100 web applications on Oct. 20, 2015. According to this figure, *jQuery-1.7.2* and *jQuery-1.10.2* are the most popular versions. Moreover, most of the famous web applications prefer to use the version 1.x.x of *jQuery* than the version 2.x.x. Figure 3.7 presents our observations for the versions of the *Modernizr* library. This figure shows that the version 2.8.3 is the preferred one, and therefore may be used preferentially.

8. http://stats.js.org/

Table 3.7 – JS.ORG rank on Oct. 20, 2015

JS.ORG rank	Library name
1	AngularJS
2	D3
3	jQuery
4	RevealJS
5	React
6	ImpressJS
7	ThreeJS
8	Backbone
9	jQueryFileUpload
10	SocketIO
...	

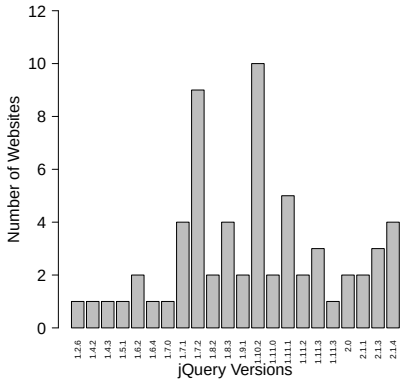


Figure 3.6 – jQuery Version Distribution on Oct. 20, 2015

3.5.3 Analysis of a Three Years Period

We used *Wayback Machine*[9], the internet archive website, to get older versions of the Alexa top 100 web applications. Thanks to *Wayback Machine*, we are therefore able to observe which libraries were used through the history of these web applications. We focused on a three years period (from Oct. 20, 2012 to Oct. 20, 2015) to observe library usage evolution. Figure 3.8 shows such observations for the top 10 used JS libraries as computed from the October 2015 snapshot. It shows that *jQuery* is the first used library for three years, and exceeds to a large extent the other libraries. The *GoogleAnalytics* library has a steep increasing slope during 20 Oct'12 and 20 Oct'13, but tends to decline since 20 Apr'15. Further, in the beginning *Modernizr* is not widely used by web applications, but accumulates popularity and exceeds *jQueryCookie* to rank third around October 2014.

As a main conclusion, Figure 3.8 shows that the usage of JS libraries evolves, but not

9. https://archive.org/web/

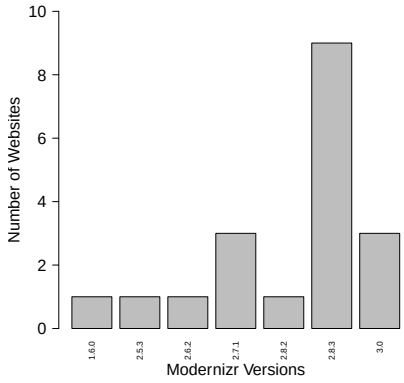


Figure 3.7 – Modernizr Version Distribution on Oct. 20, 2015

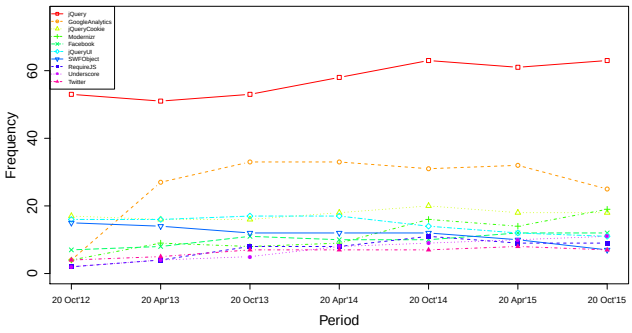


Figure 3.8 – Evolution of top 10 JS libraries for three years

that fast. During a three years period, which is quite long for web applications, there are few evolutions.

3.6 Conclusion

In this chapter we propose an approach to automatically identify JavaScript (JS) libraries used web applications. Our approach combines three different strategies that respectively aim to (1) look for any name of a library in the comment part of the JS files linked to the application, (2) compare these same linked JS files with reference files of libraries, and (3) execute a sensor JS plug-in to dynamically identify library usage.

Our approach has the advantage to be highly precise (more than 92%) once the sought libraries are included the knowledge base of our approach. We used our approach on the 100 most popular websites referenced by Alexa. We intend to identify the popular JS li-

braries, and to give some suggestions to the developers that have trouble to choose a library to include within their own web application.

Based on our observations, we can state that there are some essential libraries (such as *jQuery*, *Modernizr* or *Underscore*), and other ones that are rarely used (such as *AngularJS* and *Polymer*). This contrasts with the rankings that are provided by websites which state that *AngularJS* is very popular. We argue that our approach is focused on popular and commercial web applications, and therefore is resilient to technological buzz.

As a last result, the observations we made on three years show that the library usage evolves but not that fast. Most, if not all, the libraries that were used three years ago, are still used nowadays. As a future work, we think about partitioning web applications into several business domains to check whether some libraries are more used depending on these domains.

CHAPTER

4 How to get the standard specifications of REST services?

The REST architectural style is nowadays very popular and widely adopted by services providers. To use REST services, developers can be helped by two components: a structured REST specification and an HTML documentation. The best practice is to have a REST specification, since it can speed up the development process by generating client-side or server-side stubs, creating service compositions, etc. However, there are few available REST specifications while most of the REST services providers only provide HTML documentations. To face this issue, we describe in this chapter a semi-automated approach *ExtrateREST*, that automates the creation of REST service specifications from existing HTML documentations. Our approach inputs the index page of the HTML documentation of a REST service provider, performs an analysis of all its HTML pages, and extracts the four main parts that compose a REST specification: the base URL, the path templates, the HTTP verbs, and the associated formal parameters. *ExtrateREST* has been extensively validated to evaluate the quality of the specification it generates. It outperforms our previous approach *AutoREST* by being more effective and reliable.

Contents

4.1 Introduction

The REST architecture style defined 15 years ago by R.T. Fielding [Fielding and Taylor, 2002], is nowadays very popular and widely adopted. Many studies done either by researchers [Danielsen and Jeffrey, 2013] or by commercial sites [1] conclude that more than 75% of web services are now REST-oriented.

Curiously, as identified by Danielsen [Danielsen and Jeffrey, 2013], most of the REST APIs providers only provide a documentation of their REST API in plain HTML pages, and don't provide any structured and machine-readable specification. More precisely, according to an in-depth analysis of the 20 most popular REST services, only 20% of them provide WSDL [Chinnici et al., 2007] specifications whereas 75% provide no structured specification and only plain HTML pages [Renzel et al., 2012]. Section 2.2 gives a brief background about the REST API specifications documentations with a running example.

Building and using REST services is however challenging and, according to Renzal et al., the best practice is to have a structured specification of the REST API [Renzel et al., 2012]. Having a REST specification speeds-up the development process by enabling developers to generate client-side or server-side stubs [Fokaefs and Stroulia, 2015], or even service compositions [Wagner et al., 2012]. Additionally, a structured specification can be used to reach a better quality by inferring parameters dependency constraints [Wu et al., 2013] or performing automated tests production [López et al., 2013].

To face this situation, and mainly to help the developers of web clients that want to have REST API specifications for using existing REST APIs, we describe in this chapter an approach that automates the creation of REST API specifications from existing HTML documentations. Our approach inputs the index page of the HTML documentation of a REST API provider, performs an analysis of all its HTML pages, and generates the corresponding OpenAPI [2] specification (many specification formats exist and OpenAPI turns out to be the most popular one, with over 350,000 downloads per month). For instance, our approach has been used to generate the OpenAPI specification of Instagram (see Figure 2.3) by inputting its HTML documentation (see Figure 2.1). Section 4.3 shows the architecture of our approach.

Our work is related to the topic of web content mining [Cooley et al., 1997; Liu, 2007] but with two main specific challenges: (1) the dispersal of the information contained in the HTML pages of a REST API documentation, and (2) the heterogeneity of HTML layouts and vocabulary of the HTML documentations among the different service providers.

The challenge of the dispersal of information is due to the fact that the useful information contained in HTML documentations (the one that should be used to create the REST API specification) is commonly spread across several web pages. Furthermore, HTML documentations not only contains web pages with useful information but also web pages with

1. https://www.programmableweb.com/api-research
2. https://www.openapis.org/

useless information. As a consequence, there is a need to classify the web pages as useful or useless before to extract the information they contain [Qi and Davison, 2009; Onan, 2016].

The challenge of heterogeneity is because each of the service providers has its HTML layouts and vocabulary for its HTML documentation. For instance, some providers use HTML *TABLE* whereas others use *OL* or even *DIV* tags. As there are no common HTML layouts that are shared across all providers, machine learning approaches that aim to identify the underlying HTML layouts to extract the information are inefficient. Existing machine learning approaches, supervised [Jiménez and Corchuelo, 2016; Hogue and Karger, 2005; Sleiman and Corchuelo, 2014] and unsupervised [Velloso and Dorneles, 2017; Shi et al., 2015; Zeleny et al., 2017], give good results only when the information is contained in web pages sharing a common style, which is not true for REST API documentations.

Our approach, named *ExtrateREST*, addresses these two challenges and is a major extension of our previous work *AutoREST* [Cao et al., 2017a]. It outperforms AutoREST by being more effective and reliable (higher precision/recall). In contract to AutoREST that is fully automated and therefore performs badly on some HTML documentation depending on their HTML layouts and vocabulary, our new approach now requires a small configuration to better retrieve the information contained in HTML documentations of web service providers. This approach reaches a good trade-off between human efforts and efficiency.

Furthermore, ExtrateREST has been extensively validated to evaluate the quality of the Open API specification it generates. Our results show that it achieves a high precision/recall for both popular and randomly selected REST services than AutoREST. Section 4.4 presents the performance evaluation and discussion.

4.2 Background

This section highlights the two challenges of the extraction of a REST API specification from an HTML documentation (dispersal, heterogeneity) using a real example: the Instagram API.

4.2.1 Main challenges

The example of Instagram outlines the two challenges that should be addressed for extracting a REST API specification from an HTML documentation.

— *Dispersal*: as shown in the example, not all web pages of the HTML documentation contain useful information. For example, Figure 2.2b presents a web page to the Instagram HTML documentation that is useless. As a consequence, before to retrieve the information, useless pages should be discarded.

— *Heterogeneity*: Figure 4.1 shows the Instagram endpoint HTML layouts and vocabulary which have to be understood in order to extract the information required to

create the REST API specification. In Figure 4.1, Instagram documentation shows verb GET and URL */media/media-id* together but wraps them in separated and <code> tags. Unfortunately, the HTML layouts and vocabulary are completely different from one HTML documentation to another. For instance, Twitter wraps verb GET and URL *favorites/list* in the same <h1> tag. As a consequence, the HTML layouts and the vocabulary should be identified prior to extract the information efficiently.

```
<section class="card endpoint" >
  <header>
    <h2>
      <span class="ep-type">GET</span>
      <code>/media/</code>
      <span class="token">media-id</span>
    </h2>
  </header>
  <div class="card-info">
    .....
    <code>https://api.instagram.com/v1/media/{media-id}?access_token=ACCESS-
        TOKEN
    </code>
  </div>
  ......
</section>
```

Figure 4.1 – Code snippets of Instagram Media Endpoint in HTML documentation

4.3 ExtrateREST: an automated extractor for the generation of REST API specification

In this section, we first present the overview of our approach, named ExtrateREST, and then describe its two main steps.

4.3.1 Global architecture

Figure 4.2 presents the workflow of ExtrateREST. It inputs the index page URL of a given REST API HTML documentation and generates its corresponding OpenAPI specification. It is composed of two steps that respectively address the dispersal and heterogeneity challenges :

1. In order to deal with the dispersal of information, the first step aims at collecting the relevant pages of the REST API documentation that contain useful information. This step is done by using web crawling and machine learning algorithms. It uses crawling to gather all the pages that are directly or indirectly linked by the index page and that

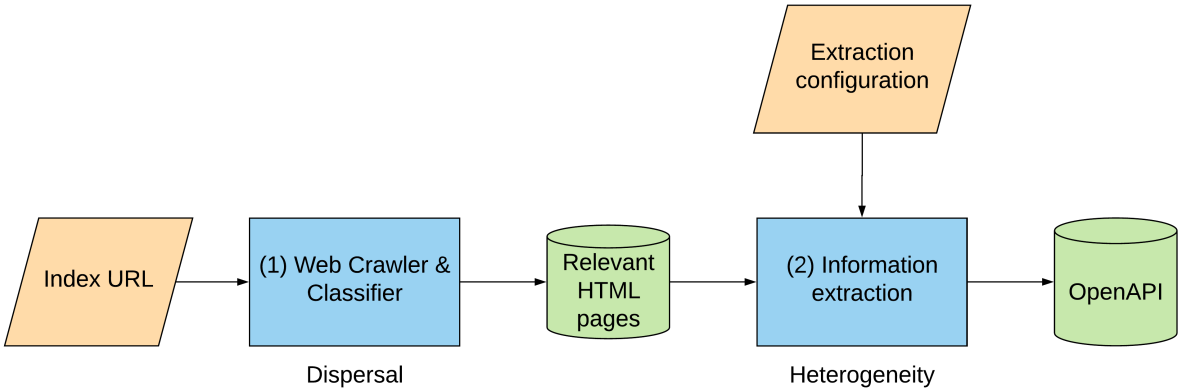


Figure 4.2 – Global workflow of ExtrateREST

belong to the same domain. It then uses machines learning algorithms to select the ones that do describe the REST API and therefore contain useful information. For instance with Instagram, this steps crawls all the pages that belong to its domain, and selects only the pages that describe a resource (see Figure 2.2a) and not the ones that just describes the platform (see Figure 2.2b).

2. The second step addresses the challenge of heterogeneity. To that extent, it runs some rules to extract the useful information, and to generate the OpenAPI specification. We predefined these rules, and each of them satisfies specific HTML layouts and vocabulary, and therefore uses dedicated mechanisms for extracting the useful information. To make these rules covering a large variety of HTML layouts and vocabulary, we manually observed 50 HTML documentations and defined a rule per any specific configuration of layouts and vocabulary. Further, we learned from our past experience that the identification of rules that apply to a given HTML documentation cannot be done by using machine learning mechanisms. We therefore design a feature model that specifies which rules should be used for a REST HTML documentation that is to be extracted. Thanks to this feature model few questions are asked to the user for letting him/her expressing the *extraction configuration* of his/her target REST HTML documentation.

The remainder of this section details each step.

4.3.2 Step 1: gather relevant HTML documentation pages

This step uses two components: a crawler and a classifier.

The crawler simply extracts all the web pages that are directly or indirectly linked by the index page of the REST API HTML Documentation. It never goes outside of the domain of the index page.

The classifier selects web pages that do contain useful information for building a REST API specification. It uses machine learning classification techniques [W.A and S.M, 2011; Koprinska et al., 2007].

To build the classifier we constructed a data set that contains HTML pages that have been manually classified as being relevant (Yes) or not (No) regarding the purpose of extracting a REST API specification. A page was said to be relevant if it contains at least one information that can be used to generate a part of a REST API specification. To build the data set, two of the authors independently looked at the web pages composing the set, and manually marked them as being relevant or not. In case of divergence between the two authors, a discussion took place and the page was marked as being useless if no consensus was obtained.

It should be noted that this data set has been inspired by our previous approach (*AutoREST*) but with two major improvements. First, we eliminated HTML tags and just considered text content, following the recommendations related to web pages content preprocessing [Qi and Davison, 2009]. Second, we extended its size and heterogeneity. Our previous data set contains web pages of the 15 most popular REST Services listed in ProgrammableWeb where popularity is defined by the number of followers. We now expand to the 30 most popular REST Services plus 10 randomly selected ones from ProgrammableWeb (see full list [3]), with a total of 1600 web pages.

Once the data set was built, we then extracted the features it contains. To that extent, each file of the data set has been treated as a plain text (one string) and transformed into a numerical feature vector by tokenizing it, counting tokens occurrences and normalizing tokens. For instance, the string "*Get a list of users who have liked this media ...*" is tokenized by using white spaces as token separators. Then, each token is assigned an integer id, such as {Get: 1, a: 2, list: 3}. Then the tokens are counted and normalized by using the TF-IDF weighting to build the feature vector [Wu et al., 2008]. We choose Random Forest [Ho, 1995] as our classification algorithm since it outperforms others in our previous work [Cao et al., 2017a].

Finally, we computed and evaluated the classifier in two dimensions: internal and external. Regarding internal performance, we consider the performance inside the data set. We split 75% of data set for the *training set* and 25% for the *testing set*, which is a default setting proposed by Pederegosa et al. [Pedregosa et al., 2011], using 4-fold cross validation. The result shows internal performance is really good with high precision (96%) and recall (96%).

Regarding external performance, we treat the whole data set as *training set* and measure the performance for a *testing set* which contains 200 pages. Those pages are selected from 10 randomly selected REST services not part of our data set. The result shows that we also achieve a reasonable precision (83%) and recall (81%) for external services.

Our classifier thus can select web pages that do contain useful information with a high

3. https://github.com/caohanyang/ExtrateREST/blob/master/APIList

precision (96%) and recall (96%) for pages in our data set, and a reasonable performance (83% precision and 81% recall) for random pages.

4.3.3 Step 2: extract information from relevant pages

This step is composed of two components: a set of pre-defined extraction rules that apply to all configurations of HTML layouts and vocabulary, and a feature model with a wizard-like questionnaire that allows users to easily identify the configurations that apply to the REST API documentation they want to target.

For building the pre-defined extraction rules, we manually observed the REST HTML documentations of 50 REST services with the intent to identify a maximum of HTML layouts and vocabulary. Those 50 REST services are composed of the 30 most popular REST services plus 20 randomly selected.

According to our observations, a REST HTML documentation has always an overview section that gives overall information about the service (e.g., base URL, authentication, rate limits). The overview section is either presented in a dedicated HTML page or in several pages with other information, depending on the service providers. A REST HTML documentation then has several REST endpoints that describe the access to the resources. Each REST endpoint is a block in one HTML page that includes a path and verb block, and optionally a parameter block and a response block (see Figure 4.3). Service providers may choose to present several REST endpoints in one page or each REST endpoint in a dedicated page.

Once we identified all the HTML layouts and vocabulary used by our samples of 50 REST HTML documentations, we then designed the extraction rules that apply to them. Technically, an extraction rule uses regular-expression, pre-processing and/or post-processing algorithms. For example there is a rule that extracts the *Parameter Type* field if it is encoded by an HTML *Table* element. Another rule extracts the same field if it is encoded by an HTML *List*, etc. Some rules are more complex, such as the one that extracts the *Base Url* field when it is encoded as a prefix URL in all URLs that exist in the endpoints. This rule fetches all of URL encoded in the endpoints and extracts their largest common prefix. For the sake of simplicity, we don't present all our extraction rules in this chapter. Figure 4.3 highlights three rules that extract useful information for the *Base URL*, *Path and Verb* and *Parameters* fields.

Having all extraction rules is mandatory for extracting useful information. However, when a user wants to target a given HTML documentation, he/she first has to identify the ones that really apply to its layouts and vocabulary. To that extent, we learned from our past experience that machine learning algorithms are not efficient in this setting [Cao et al., 2017a]. Existing machine learning approaches, supervised [Jiménez and Corchuelo, 2016; Hogue and Karger, 2005; Sleiman and Corchuelo, 2014] and unsupervised [Velloso and Dorneles, 2017; Shi et al., 2015; Zeleny et al., 2017] are not appropriate for identifying which rules optimally match a given HTML documentation.

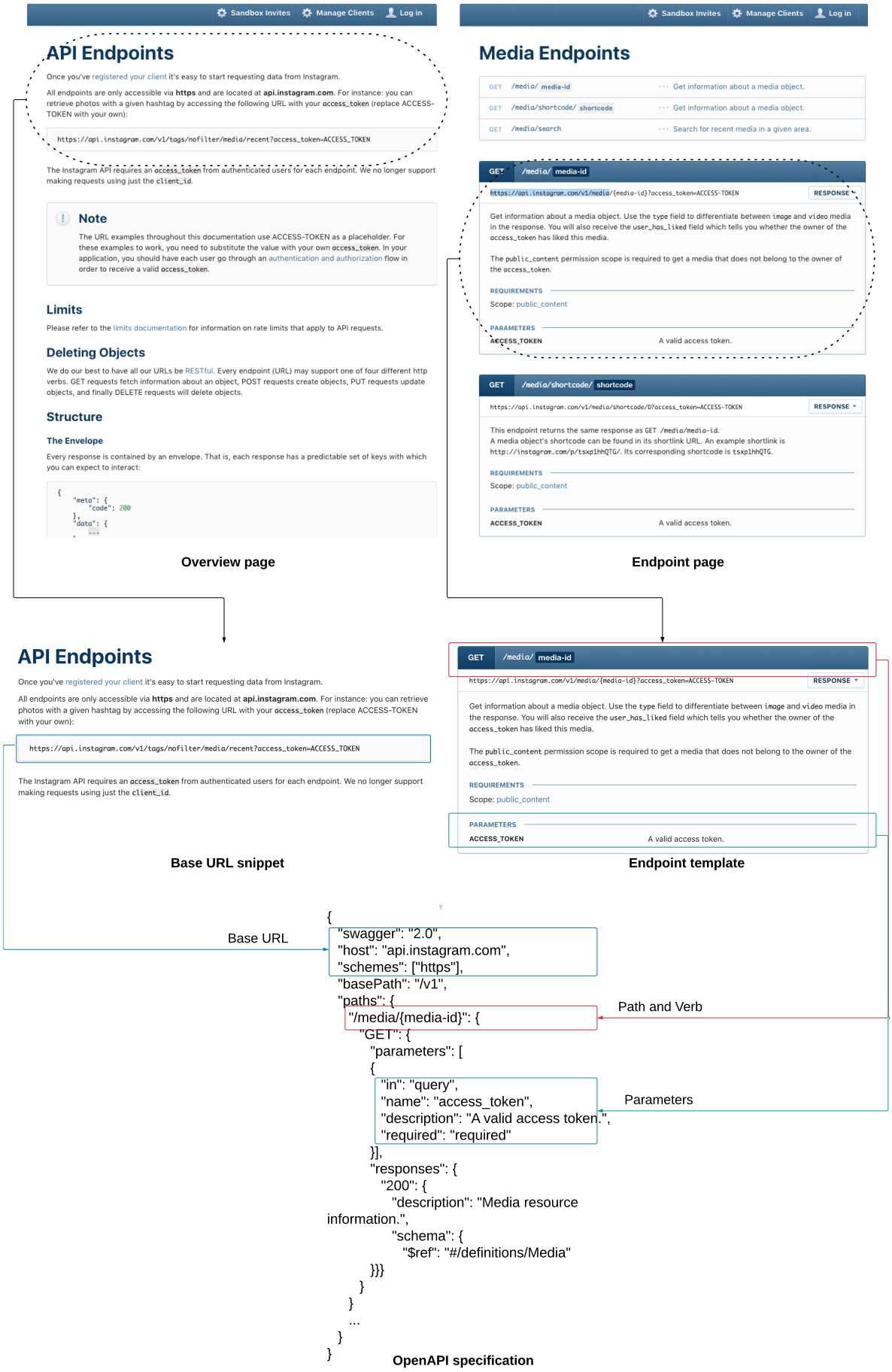


Figure 4.3 – The information extractor to build specification.

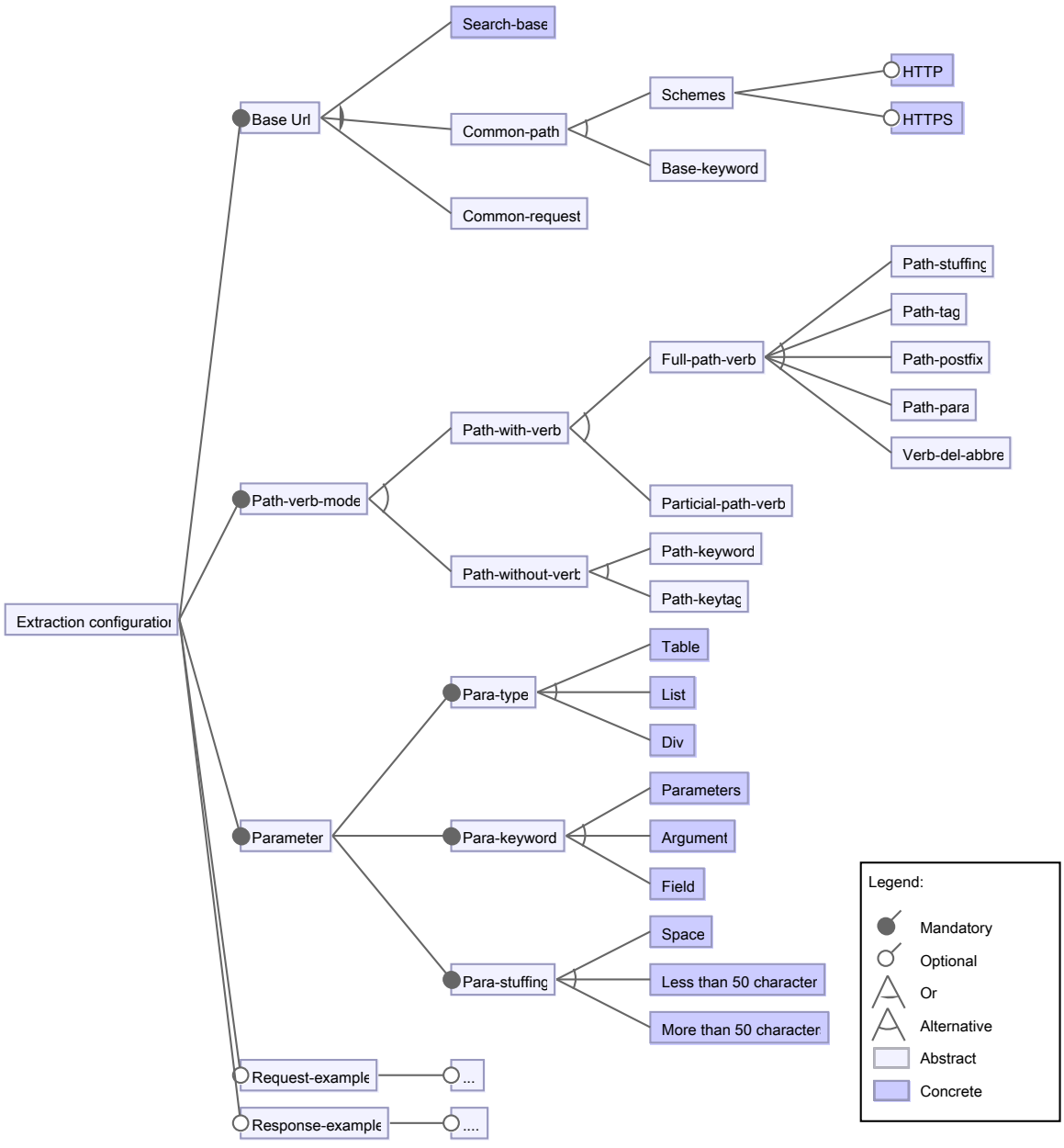


Figure 4.4 – Feature model for the extraction configuration (partial).

To face this problem of rules identification, we organized all our rules in a feature model (see Figure 4.4 for an extract). Each feature of this model expresses a configuration of rules. The features are decomposed hierarchically, and marked as mandatory or optional if they target mandatory or optional information. For instance, the *Base Url* is decomposed of three sub-features that describe three possible configurations of rules to extract the *Base Url* field. Further, the *Base Url*, *Path-verb-mode* and *Parameter* features are mandatory as they identify the configuration of rules used to build the *Base Url*, *Path Templates*, *Verbs* and *Paramaters* mandatory fields of the REST API Specification. In opposite, the *Request-example* and *Response-example* are optional.

Figure 4.4 presents a small extract of our feature model. This extract exhibits the complexity of all the configurations of rules, and reflects the diversity of all layouts and vocabulary that are used by REST documentations. Our feature model contains more than 7000 choices for configurations.

Our feature model is so complex that brute force cannot be performed to identify which set of configurations should be used to optimally extract the useful information of a given HTML documentation. For large REST services that involve hundreds of endpoints, using brute force and testing each of them is too expensive in time. For example, Facebook includes 306 endpoints and our approach require 2 minutes to extract a specification with a given configuration. It would then require 9.72 days (14000 minutes) to try all possible choices of configurations.

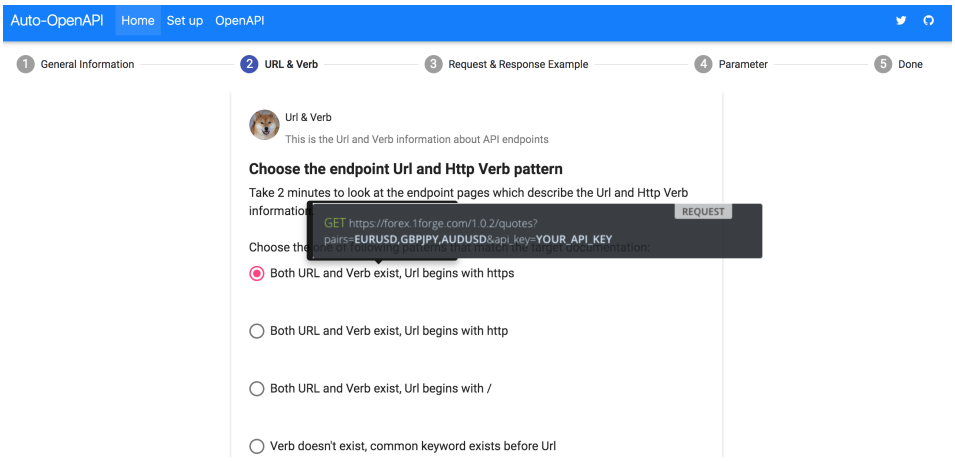
To ease the identification of configurations, we therefore designed a wizard-like questionnaire composed of twelve simple questions that supports users to identify the key features that best match the HTML documentation they want to target. Our wizard-questionnaire [4] comes with a web interface that guides developers step by step through all the questions (see Figure 4.5a).

4.4 Evaluation

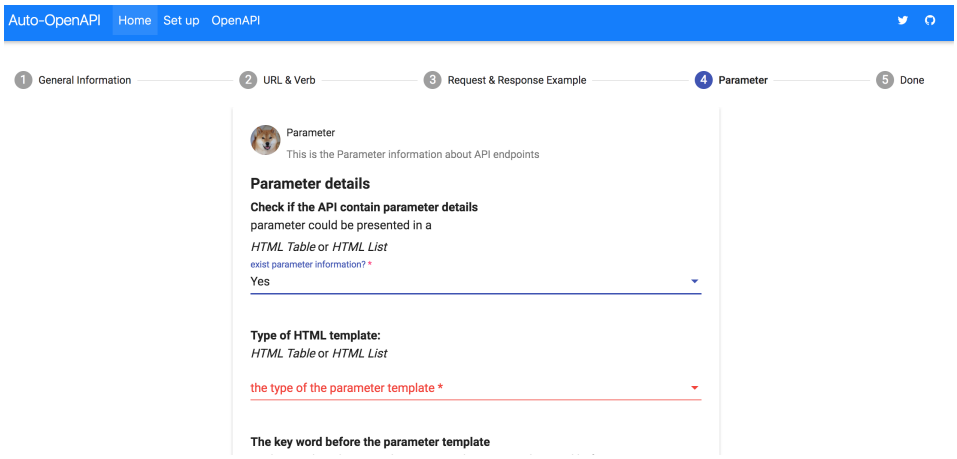
ExtrateREST has been developed in Python and Java, and is available on-line as an open source project [5]. This section presents its evaluation. It begins by presenting the quantitative evaluation, and finishes with a discussion including observations explaining the limits of our approach.

In our quantitative performance evaluation we measure the four mandatory parts of a REST specification: Base URL, Path Templates, Verbs, and Parameters. The goal is to reflect the quality of the generated REST specification regarding these parts. We measure the quality of a generated REST specification according to the following criteria:

4. http://extraterest.ml/
5. https://github.com/caohanyang/ExtrateREST



(a) ExtrateREST global rule page screenshot



(b) ExtrateREST parameter page screenshot

Figure 4.5 – Two screenshots for ExtrateREST front-end

— The quality of the Base URL is measured by a boolean. True means that the specification exactly reflects what is written in the documentation. False means that the Base URL does not reflect the documentation.

— The quality of the Path Templates is measured by counting the number of Paths templates in the specification and in the documentation, and by checking how much of them match. The quality is then expressed with precision (No. Match/No. in Spec.) and recall (No. Match/No. in Doc.).

— The quality of the Verbs is measured by counting the number of Verbs in the specification and in the documentation, and by checking how much of them match. Two verbs match if they have the same Path Template and if they are the same. The quality

Table 4.1 – Quantitative Comparison for topmost popular and random REST service

	ExtrateREST		**Outperform AutoREST ratio**	
	precision	recall	precision	recall
Most popular service				
Base URL	80.0%	80.0%	100%	100%
Path Templates	93.9%	93.2%	100%	90.9%
Verbs	96%	92.7%	100%	100%
Parameters	75.5%	71.0%	100%	100%
Randomly selected service				
Base URL	86.7%	86.7%	100%	1000%
Path Templates	87.7%	82.9%	90%	70%
Verbs	84.5%	82.3%	90%	90%
Parameters	58.8%	51.6%	100%	100%

is then expressed with precision (No. Match/No. in Spec.) and recall (No. Match/No. in Doc.).

— The quality of the Parameters is measured by counting the number of Parameters in the specification and in the documentation, and by checking how much of them match. Two parameters match if they have the same Path Template, the same verb, the same name and the same type. The quality is then expressed with precision (No. Match/No. in Spec.) and recall (No. Match/No. in Doc.).

We evaluate ExtrateREST on two corpora of REST services (see full corpora [6]). The first corpus is composed of the 15 most popular REST services. The second corpus is composed of 15 REST services selected at random from ProgrammableWeb [7]. Those randomly selected services are different from 50 services that used to build the feature model.

The first author did use the questionnaire (see Section 4.3.3) to identify each of the extraction rules that 30 REST services. It took in average five minutes to answer all the questions. Once generated, all the quality criteria were measured manually by comparing the generated specification with its corresponding documentation.

The evaluation done with the first corpus shows how ExtrateREST performs on popular REST services knowing that it has been trained using some of them (see Section 4.3). The evaluation done with the second corpus expresses the capacity of ExtrateREST to generate OpenAPI specifications for random services

As a main result, except for *parameters* with random REST services, ExtrateREST performs well (see Table 4.1). As shown in the results, ExtrateREST is quite good for generating

6. https://github.com/caohanyang/ExtrateREST/tree/master/ExtrateREST_app/ExtrateREST_core/Corpus

7. https://www.programmableweb.com

a Base URL. We observed that it fails when the overview page includes the Base URL, but also many other useless information, which makes it filtered out by our classifier. For example, it fails with Twilio whose overview page is not classified as an interesting page. It also fails when REST HTML documentation is rendered dynamically thanks to JavaScript. For instance, it fails with Wikipedia whose HTML DOM is generated by JavaScript functions and thus is hard to crawl.

Figure 4.6 and 4.7 then present the precision and recall for the Path Templates, Verbs and Parameters, respectively. It should be noted that when ExtrateREST fails in finding the Base URL, it also fails for all of the other parts. For Path Templates and Verbs, ExtrateREST achieves a good result for both most popular corpus (average precision and recall are more than 90%) and for randomly selected corpus (average precision and recall are more than 85%). Furthermore, when compares the two corpora, it is clear that ExtrateREST performs better for popular REST services, as it has been trained on it. After the manual investigation, we found ExtrateREST fails mainly for two reasons. First, it uses a regular expression to detect URLs but as there are many URLs in web pages it sometimes fails to distinguish the ones that correspond to REST services. Second, service providers do not always use the same templates to present all endpoints, even if it is the case for all the REST documentation of our training corpus. Hence our ExtrateREST can the extract the main templates but not all. That's one of the reasons why ExtrateREST performs worse in random REST services since their documentations are less consistent. Finally, ExtrateREST is good to extract the Parameters for popular REST services but not for the ones that have been randomly selected. The main reason is that the documentation provided by the latter is not structured with tables or lists, as it is expected by our information extractor.

We further made a comparison with our previous approach *AutoREST*. To that extend, we performed the same evaluation on the same corpora but by using AutoREST instead of ExtrateREST. The right part of Table 4.1 presents this comparison. For finding the Base URL, ExtrateREST obtains better results both on the most popular REST services (12/15 in contrast to 11/15 for AutoREST), and on randomly selected REST services (13/15 in contrast to 10/15 for AutoREST). In order to make a fair comparison, we only calculated precisions and recalls for the 11 most popular and 10 randomly selected REST services whose Base URL is true for both approaches. For the Path templates of most popular services, ExtrateREST achieves an average 93.9% precision and 93.2% recall. Furthermore, ExtrateREST performs equal or better than AutoREST 11 times in the precision and 10 times in the recall. The improvement is mainly due to the specific extraction configuration for each REST service, in order to solve the challenge of heterogeneity, by adding human efforts. As a conclusion, ExtrateREST has largely improved the performance of Path Templates, Verbs, and Parameters, especially for randomly selected services.

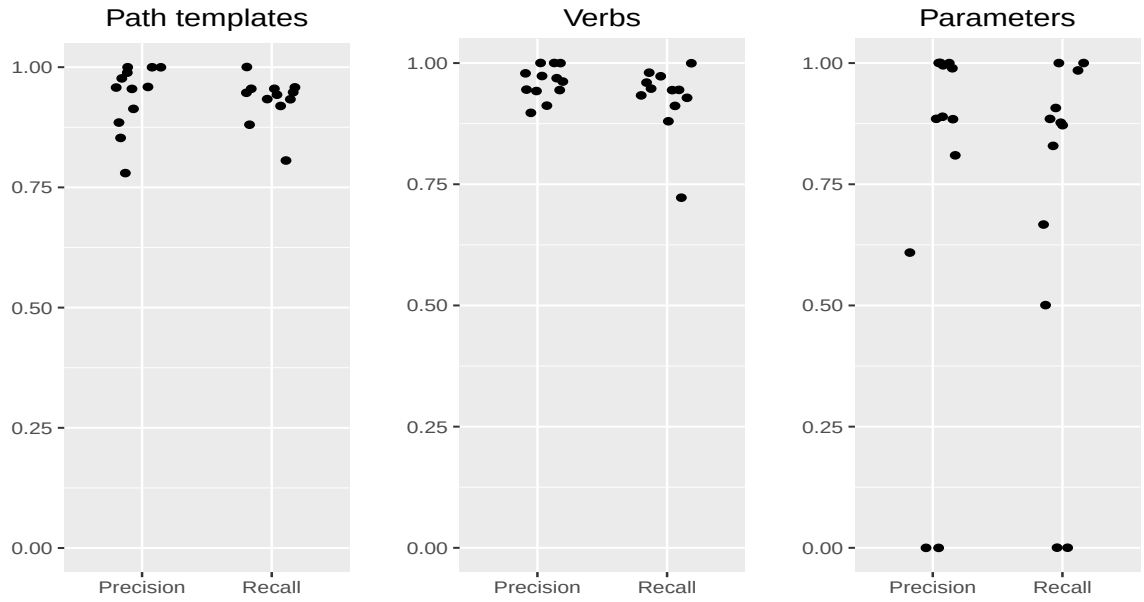


Figure 4.6 – Results on the most popular REST Services

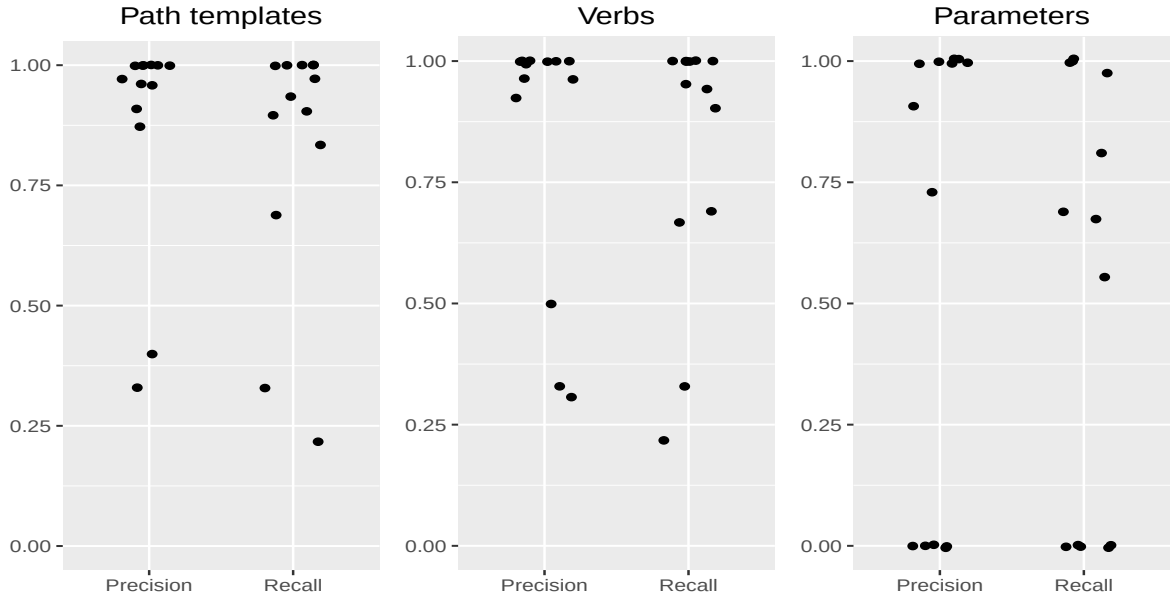


Figure 4.7 – Results on randomly selected REST Services

4.5 Conclusion

REST API specifications provide many useful facilities to developers. However, many REST services are still only documented by plain HTML pages, and don't provide such specifications. In this chapter we then present ExtrateREST, an approach for automatically transforming an HTML documentation into an OpenAPI specification.

ExtrateREST inputs a single index URL, gets all the documentation pages that are linked by it, selects the useful ones thanks to a machine-learning algorithm, extracts the endpoint information according to a manually-built extraction configuration, and then produces the corresponding OpenAPI specification. It is fully available as an open-source project.

ExtrateREST solves two main challenges. The first one is the dispersal of the information contained in the HTML pages of a REST API documentation. It settles this challenge by classifying the HTML pages and selecting interesting ones. The second challenge is the heterogeneity of HTML layouts and vocabulary of the HTML documentations among the different service providers. It then resolves this challenge by using an extraction configuration provided by developers.

The validation we performed shows that ExtrateREST produces good results for popular REST services as well as randomly selected ones. ExtrateREST returns a partial but quite complete specification for four-fifths of the popular REST services. For randomly selected REST services it is less successful mainly because the provided HTML documentation is not structured as one of the most popular REST services.

As future work, we are exploring how to use real API calls to enhance the specification produced by ExtrateREST.

CHAPTER

5 How to adapt to the data changes of REST services?

REST APIs together with JSON are commonly used by modern web applications to export their services. However, these services are usually reachable in a pull mode which is not suitable for accessing changing data. Turning a service from a pull to a push mode is therefore frequently asked by web developers that want to get notified of changes. Converting a pull API into a push one obviously requires to make periodical calls to the API but also to create a patch between each successive version of the data. The latter is the most difficult part and this is where existing solutions have some imperfections. To face this issue, we present a new patch algorithm supporting *move* and *copy* change operations. Our evaluation done with real industrial data shows that our algorithm creates small patches compared with other libraries, and creates them faster.

Contents

5.1 Introduction

Most of the web applications [1] provide an access to their services thanks to a REST API [Fielding and Taylor, 2002]. Their services are then directly reachable by HTTP requests, where the exchange of data is commonly done in JSON, the JavaScript Object Notation [Crockford, 2006].

Converting a pull mode API into a push mode one obviously requires to make periodical calls to the API but also to create a patch between each successive received versions of the data. The latter is the most difficult part and this is where existing solutions have some imperfections. Indeed, creating a patch between two documents is a well-known very complex problem [Zhang and Shasha, 1989; Buttler, 2004], which has not been studied yet for JSON documents. A JSON document is a labeled unordered tree that contains arrays (ordered sequences). Creating a patch between two JSON documents may therefore lead to an NP-hard problem depending both on the change operations that are considered (add, remove, move, copy), and on the quality of the created patch (in terms of size).

In this chapter we propose a new patch algorithm that is tailored to JSON documents, and that drastically improves the conversion of pull mode APIs into push mode ones. Our algorithm returns a JSON Patch as specified by the JSON Patch RFC [Bryan and Nottingham, 2013]. It therefore handles any changes that can be done on JSON documents, either on their basic properties or their arrays, and supports simple changes (add, remove) as well as complex ones (move, copy), which allows clients to deeply understand changes that have been done.

We implemented our algorithm in JavaScript as it is the most common language used in web applications. We validate it by making a comparison with other JavaScript libraries that support the JSON patch RFC. This validation has been done by using real data provided by our partner StreamData.io.

As the main result, we provide:

— A new JSON patch algorithm that fully complies with the JSON Patch RFC.

— A JavaScript implementation of our algorithm that performs better than the existing ones.

The structure of the chapter is as follows. The Section 5.2 presents our algorithm (named JDR). The Section 5.3 then presents the evaluation of our JavaScript framework implementing our algorithm. The Section 5.4 finally presents our conclusion.

5.2 JDR: a JSON patch algorithm

By running several APIs we observed that changes performed to JSON documents commonly target a complete sub-tree, but never target several internal nodes. More precisely,

1. https://www.publicapis.com/

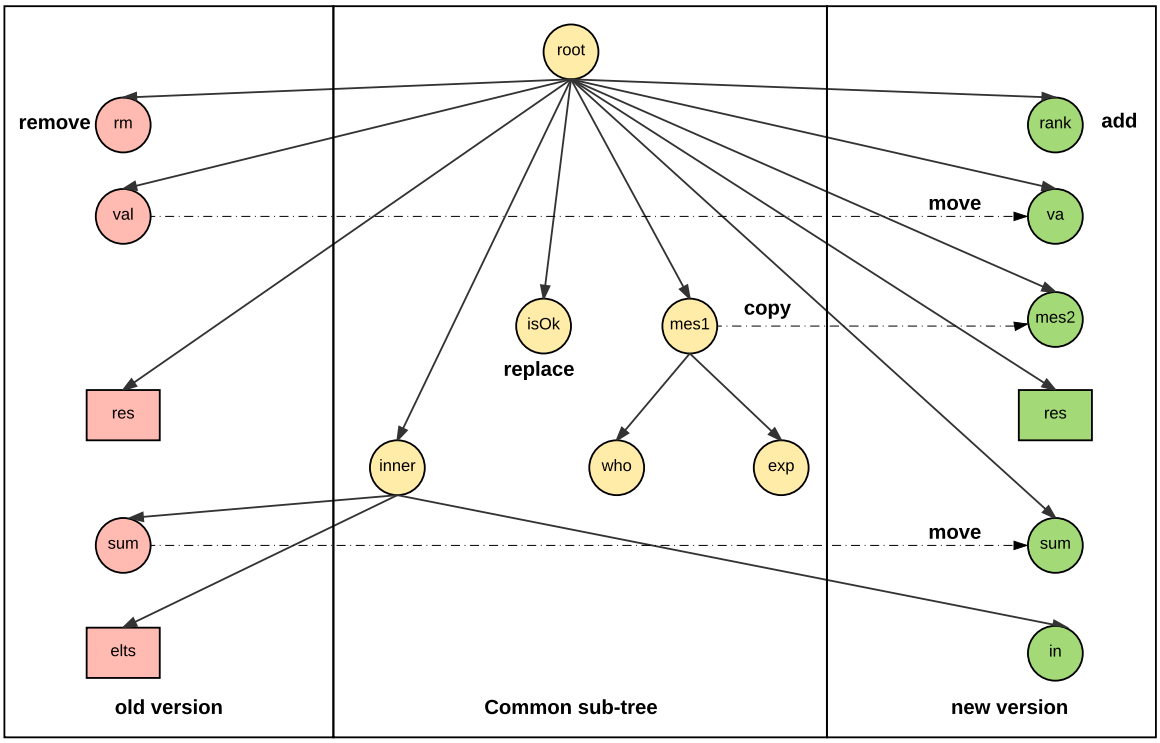


Figure 5.1 – The two versions of our example as a tree with object and label node presented with circles and array nodes with square. The central part represents the common sub-tree. The left part presents nodes direct children of the common tree and that belong to the *old* version. The right part presents nodes direct children of the common tree and that belong to the *new* version.

a change either adds, removes, replaces, moves or copies a complete sub-trees but never changes the topology of a sub-tree by inserting, removing, or moving some nodes inside the sub-tree. The same is true for arrays, changes made to arrays always target one array but they never target two or more different arrays. These observations have then driven the design of our algorithm that aims to identify large sub-trees or arrays, which are targets of changes.

Based on this consideration, our algorithm inputs two versions of a JSON document (the *old* and the *new* versions) and proceeds the three following steps:

Step 1: build common sub-tree. First it builds a large common sub-tree that is shared between the *old* and the *new* versions. This sub-tree contains the root node of both the *old* and *new* versions, and all the object and literal nodes that both exist in the *old* and *new* versions, in the same locations, with the same labels (values can be different). The

array nodes are considered in the following steps. The center part of the Figure 5.1 presents the common sub-tree for our example. Once the common sub-tree has been created, for each of its label leaf node, if the value is not the same in the old and new version, a $replace$ operation with the value of the new version is put into the patch. With our example, the $isOk$ node corresponds to such a case.

Algorithm 1 shows the pseudo code of this step. We assume that nodes have a `kind` attribute which has a value in the {literal, object, array} set. We also assume that nodes have an `hash` attribute that corresponds to the hash of the subtree rooted at this object for object nodes, the hash of the value for literal nodes, and the hash of the sequence for array nodes. They also have a `props` attribute containing the set of properties owned by the object. Object nodes and array nodes have a `path` attribute that contains their JSON path. JSON Literal have `type` attribute which has a value in the {boolean, number, string} set, and a `value` attribute that contains the literal's value. In this step, we use a $diff()$ function to build the tree presented in Figure 5.1. It generates the hashmap $commons$ for the large common sub-tree, hashmap $deletions$ for the left part sub-tree and hashmap $additions$ for the right part sub-tree. In Line 6, the $deepEqual()$ function recursively compares two JSON nodes and return true only when they are completely equal. In Line 28, the $compareArray()$ function compares two JSON arrays and will be discussed in Step 3.

Step 2: compare two sub-trees. Second, for each object or literal node of the old version that does not belong to the common sub-tree but whose direct parent belongs to it, put a remove operation in the patch and mark the node as a removed one, unless there is a marked added node with the same value. In that case, put a move operation in the patch and mark the node as a moved one. The left part of the Figure 5.1 presents these nodes. The rm node is a removed node. The val and sum nodes are moved nodes. Symmetrically, for each node of the new version that does not belong to the common sub-tree but whose direct parent belongs to it, put an add operation in the patch and mark the node as an added one, unless there is a marked removed node or a node in the common sub-tree with the same value. In case of a removed node, put a move operation in the patch and mark the node as a moved one. In case of a node in the common sub-tree, put a copy operation in the patch and mark the node as a copied node. The right part of the Figure 5.1 presents these nodes for our example. The $rank$ and in nodes are added nodes. The val and sum nodes are moved nodes. The $mes2$ node is a copied node.

Algorithm 2 shows the pseudo code of this step. With input hashmaps (i.e., $commons$, $additions$, $deletions$), it loops nodes in $deletions$ and compares with ones in $additions$, with purpose of creating related patch actions. In Line 21, the $findCommonNodeByHash()$ function will search the current node in common sub-tree, if found node with same hash value, it can generate the $copy$ action.

Algorithm 1 The diff algorithm

```
1: commons = hashmap()
2: additions = hashmap()
3: deletions = hashmap()
4: actions = list()
5: function DIFF(old, new, actions)
6:     if deepEqual(old, new) then                    // compare two JSON nodes recursively
7:         commons.add[old.hash].add(old)
8:         return
9:     end if
10:    if old.kind ≠ new.kind then
11:        deletions[old.hash].add(old)
12:        additions[new.hash].add(new)
13:    else if old.kind == literal then               // literal node
14:        if old.value ≠ new.value then
15:            actions.add(replace(old.path, new.value))
16:        end if
17:    else if old.kind == object then                // object node
18:        for all prop ∈ old.props ∩ new.props do
19:            compare(old[prop], new[prop], actions)
20:        end for
21:        for all prop ∈ old.props \ new.props do
22:            deletions[old[prop].hash].add(old[prop])
23:        end for
24:        for all prop ∈ new.props \ old.props do
25:            additions[new[prop].hash].add(new[prop])
26:        end for
27:    else if old.kind == array then                 // array node
28:        compareArray(old, new, actions)            // compare two JSON arrays
29:    end if
30: end function
```

Algorithm 2 The compare algorithm

```
1: commons = hashmap()
2: additions = hashmap()
3: deletions = hashmap()
4: actions = list()
5: function COMPARE(old, new, actions)
6:     // iterate keys in the deletions hashmap
7:     for all key ∈ deletions.keys do
8:         delNodes = deletions[key]
9:         // compare keys in the additions hashmap
10:        for all addNode ∈ additions[key] do
11:            if delNodes.size() > 0 then
12:                delNode = delNodes.pop()
13:                // move or delete operation
14:                if delNode.hash == addNode.hash then
15:                    actions.add(move(delNode.path, addNode.path))
16:                else
17:                    actions.add(deletion(delNode.path))
18:                end if
19:            else
20:                // search current node in common sub-tree
21:                node = findCommonNodeByHash(commons, addNode)
22:                // copy or add operation
23:                if node ≠ NULL then
24:                    actions.add(copy(node.path, addNode.path))
25:                else
26:                    actions.add(addition(addNode.path, addNode.json))
27:                end if
28:            end if
29:        end for
30:    end for
31: end function
```

Step 3: compare JSON array. Third, for each array node in the old version whose direct parent belongs to the common sub-tree, check if there is an array node in the new version, child of the same parent and with the same label. If so compare the two arrays (see the following array algorithm). If not, put a remove operation in the patch. For each array node in the new version whose direct parent belongs to the common sub-tree, put an add operation in the patch. The Figure 5.1 presents these nodes. The res nodes are then compared. The $elts$ node is removed.

As just described, our algorithm only creates a patch for two versions of a same array if and only if the array is in the exact same location in the two versions of the JSON document. Further as the change operations defined by the RFC can only target cells one by one (it is not possible to remove or move several cells with one operation), there is then no need to compute a LCS (Longest Common Subsequence [Hirschberg, 1977]). Comparing pairs of cells is therefore sufficient for creating a patch.

The creation of the patch is then done by comparing the cells of the array with the intent to identify the common ones, the ones that have been removed and the ones that have been added. More precisely, our algorithm first sorts the cells of the two versions of the array by computing a similarity hash [2] of their value. Secondly, thanks to the similarity hash order, it iterates through the cells in the two versions of the array and creates a temporary array patch by applying the following rules. If an old cell has a corresponding new cell (with the same value), a move operation is put into a temporary patch. If an old cell has no corresponding new cell, a remove operation is put into the patch. If a new cell has no corresponding old cell, an add operation is put into the patch. The following pseudo code shows the part 3.

Step 4: transform patch array. Fourth, it transforms the temporary array patch into a final patch by taking care of the indexes of the changed cells because the execution of a change operation may have an impact on the indexes of the following ones. This transforming index method is inspired by the classical Operational Transformation (OT) technology, which aims to solve concurrency control of collaborative editing in distributed systems [Ellis and Gibbs, 1989; Lamport, 1978]. To that extent, it sorts the operations of the temporary patch according to the indexes of the changed cells and to the type of change ($move < remove < add$), iterates through them and recompute the indexes. If a $move$ operation moves a cell to the same operation (the target index is equal to the source index), it is removed from the patch. Furthermore, we would optimize the patch by adding $replace$ and $copy$ operations during the iteration. A add operation can be converted to $replace$ when the previous operation is $delete$ with the same target index. A add operation can be transformed into $copy$ when cells with same value can be founded in the common part of the two arrays. This step is complex and we shows the pseudo code in Algorithm 4 and 5 .

2. https://github.com/darkskyapp/string-hash

Algorithm 3 The compareArray algorithm

```
1: function COMPAREARRAY(old, new, actions)
2:     oldPos = hashmap()
3:     newPos = hashmap()
4:     tmpPatch = list()
5:     // build new arrays with hash order
6:     for i = 0 → i < old.size do
7:         oldPos[old[i].hash].add(i)
8:     end for
9:     for i = 0 → i < new.size do
10:        newPos[new[i].hash].add(i)
11:    end for
12:    // iterate cells owned by both arrays
13:    for all key ∈ oldPos.keys ∩ newPos.keys do
14:        for i = 0 → i < max(oldPos[key], newPos[key]) do
15:            if i ≥ oldPos[key].size then
16:                tmpPatch.add(arrayAddition(newPos[key][i], new[newPos[key][i]]))
17:            else if i ≥ newPos[key].size then
18:                tmpPatch.add(arrayDeletion(oldPos[key][i]))
19:            else
20:                tmpPatch.add(arrayMove(oldPos[key][i], newPos[key][i]))
21:            end if
22:        end for
23:    end for
24:    // cells owned solely by old array should be delete
25:    for all key ∈ oldPos.keys \ newPos.keys do
26:        for all i ∈ oldPos[key] do
27:            tmpPatch.add(arrayDeletion(i))
28:        end for
29:    end for
30:    // cells owned solely by new array should be add
31:    for all key ∈ newPos.keys \ oldPos.keys do
32:        for all i ∈ newPos[key] do
33:            tmpPatch.add(arrayAddition(i, new[i])
34:        end for
35:    end for
36: end function
```

Algorithm 4 The transformPatch algorithm

```
1: tmpPatch = list()
2: actions = list()
3: arrCommon = list()
4: function TRANSFORMPATCH(tmpPatch, actions)
5:     // sort the temporary patch according to the indexes and operation type
6:     sort(tmpPatch)
7:     for i = 0 → i ≤ tmpPatch.size do
8:         action = tmpPatch[i]
9:         if action.type == move then
10:            action.from = changeIndex(action, i, tmpPatch)
11:            if action.from == action.to then
12:                // remove the action
13:                arrCommon.add(action)
14:                tmpPatch.delete(action)
15:                continue
16:            else
17:                // update the move action with new index
18:                tmpPatch.update(action)                    // the move operation
19:            end if
20:        else if action.type == deletion then
21:            action.at = changeIndex(action, i, tmpPatch)
22:            tmpPatch.update(action)                        // the delete operation
23:        elseaction.type == addition
24:            action.at = changeIndex(action, i, tmpPatch)
25:            if tmpPatch[i − 1].type == deletion & action.at == tmpPatch[i − 1].at
    then                                                       // the replace operation
26:                tmpPatch.delete(tmpPatch[i − 1])
27:                action.type = replace
28:                tmpPatch.update(action)
29:            else if copyIndex = findCopyIndex(action, i, tmpPatch, arrCommon) ≠
    −1 then                                                    // the copy operation
30:                action.type = copy
31:                action.from = copyIndex
32:                tmpPatch.update(action)
33:            else                                           // the add operation
34:                tmpPatch.update(action)
35:            end if
36:        end if
37:    end for
38:    actions.add(tmpPatch)
39: end function
```

Algorithm 5 The changeIndex algorithm

```
1: function CHANGEINDEX(action, m, tmpPatch)
2:     var index
3:     if action.type ∈ {add, replace, copy} then
4:         // the indexes of those actions don't change
5:         return action.at
6:     else if action.type == deletion then
7:         index = action.at
8:     else if action.type == move then
9:         index = action.from
10:    end if
11:    // calculate the impact of previous actions on the index
12:    for i = 0 → m ≤ tmpPatch.size do
13:        preAction = tmpPatch[i]
14:        switch preAction.type do
15:            case remove
16:                if index > preAction.at then
17:                    index−−              // reduce index since the previous deletion
18:                end if
19:            case add, copy
20:                if index >= preAction.at then
21:                    index++              // increase index since the previous addition
22:                end if
23:            case replace                 // no impact from the previous replacement
24:            case move
25:                min = Math.min(preAction.from, preAction.to)
26:                max = Math.max(preAction.from, preAction.to)
27:                // only have impact when index in the move (from, to) section
28:                if index ∈ (min, max) then
29:                    if preAction.from > preAction.to then
30:                        // increase index when previous action move to the front part
31:                        index++
32:                    else
33:                        // reduce index when previous action move to the back part
34:                        index−−
35:                    end if
36:                end if
37:    end for
38:    return index
39: end function
```

```
[
  { "op": "add",     "path": "/rank",  "value": 6 },
  { "op": "remove",  "path": "/rm"},
  { "op": "replace", "path": "/isOk",  "value": false},
  { "op": "move",    "path": "/va",    "from": "/val"},
  { "op": "copy",    "path": "/mes2",  "from": "/mes1"}
  { "op": "add",     "path": "/res/0", "value": "v5"},
  { "op": "replace", "path": "/res/2", "value": "m2"},
  { "op": "remove",  "path": "/res/4"},
  { "op": "copy",    "path": "/res/3", "from": "/result/1"},
  { "op": "move",    "path": "/res/5", "from": "/result/4"},
  { "op": "remove",  "path": "/inner/in/elts"},
  { "op": "add",     "path": "/inner/in", "value": {"elts":["a", "b", "c
      "]}},
  { "op": "move",    "path": "/sum",   "from": "/inner/sum"}
]
```

Figure 5.2 – A RFC JSON Patch generated by our approach that, if applied to *source* JSON document of the Figure 2.10, would get the *target* JSON document.

The Figure 5.2 finally presents the patch created by our approach. The main difference with an optimal patch is that nodes that are not direct children of the common sub-tree are not target of any change. With our example, the sub-tree with the node in as a root is therefore fully created by the patch, and its child node $elts$ is created from scratch whereas it should have been moved.

5.3 Efficiency evaluation

Our patch algorithm has been developed in JavaScript and is available as an Open Source library.[3] We present in this section its efficiency evaluation in comparison with all other existing JavaScript libraries that support the JSON Patch RFC (see Table 2.3).

Our evaluation consists in asking all the libraries to create JSON patches. We then compare them according to two quantitative factors: the time required to create the patch, and the size of the patch. Our claim is that a library is considered to be efficient if the patches it creates are small and if it creates them quickly.

Our evaluation is fully automated. It inputs a given REST service that provides access to a changing data, and periodically calls it 61 times to get 61 different versions of the changing data. Then, for each of the 60 consecutive versions it asks to all the existing libraries to generate the corresponding patch, and compares the time they take as well as the size of their returned patch. We repeat the generation of the patch 100 times to get an average value for both time and size. Our evaluation then returns 60 average values for both time and size for each library and for any given REST service. The evaluation has been executed

3. https://github.com/caohanyang/json_diff_rfc6902

on a desktop computer Intel Core i7-4770 CPU @3.40GHz×8, 16GB of RAM, and Ubuntu 14.04.2 LTS x86 64.

The choice of the called REST service has obviously an impact on the results obtained by our evaluation. We therefore choose to include into our dataset only real services provided by well-known web applications. Further, as the existing libraries mainly differ by their support of changes (see Table 2.3), we choose to include into our dataset three kinds of services: the one where changes are only made to objects' properties, the one where changes are only made to arrays, and the one where changes are made to both. Our industrial partner Streamdata.io then provides us one service for each such kinds. Our dataset, available on GitHub [4], includes the *Xignite* GetRealTimeRate, *Stackoverflow* Answers and *Twitter* Timeline services.

The Xignite GetRealTimeRate [5] service provides real-time currencies in the global financial market. The service returns a JSON document that contains one node object for each of the selected currencies (i.e. EURUSD, USDGBP). Changes between two successive versions are then only made to the properties of these objects. It should be note that a period of 15 seconds has been advised by our industrial partner between two consecutive versions.

The Stackoverflow Answers [6] service provides a list of Stackoverflow's answers . The service returns a JSON document that contains an array with the latest 20 answers. Changes between two successive versions are then only made to the array (new answers are added, last ones are deleted).

The Twitter Timeline [7] service provides the home timeline of a specific account with up-to-date Tweets. The service returns a collection of the most recent 20 Tweets of the authenticated user. Changes between two successive versions can be made to the array or to the objects themselves when tweets' properties change.

The Figure 5.3 shows an extract of successive versions that have been obtained by calling the services of our dataset. It clearly shows that changes performed to the data can be done either on objects' properties with the Xignite service, or on the array with Stackoverflow, or on both with Twitter.

The Figure 5.4, 5.5, 5.6 then present the results of our evaluation for each service. Each figure presents one figure for the time and one figure for the size where the black dots present the 60 average values, and the bold red dot presents the median of these average values.

For the Xignite service, Fast-JSON-Patch, JDR and rfc6902 always generate small patches while jiff doesn't (see Figure 5.4a). Curiously JSON8 chooses to simply replace the whole JSON document. Regarding time, Fast-JSON-Patch is the fastest followed by our

4. https://github.com/caohanyang/json_diff_rfc6902/tree/master/dataset
5. http://globalcurrencies.xignite.com/xGlobalCurrencies.json/GetRealTimeRate?Symbol=EURUSD,USDGBP,EURJPY,CHFDKK&_token=[YOUR_TOKEN]
6. https://api.stackexchange.com/2.2/answers?order=desc&sort=activity&site=stackoverflow
7. https://api.twitter.com/1.1/statuses/home_timeline.json

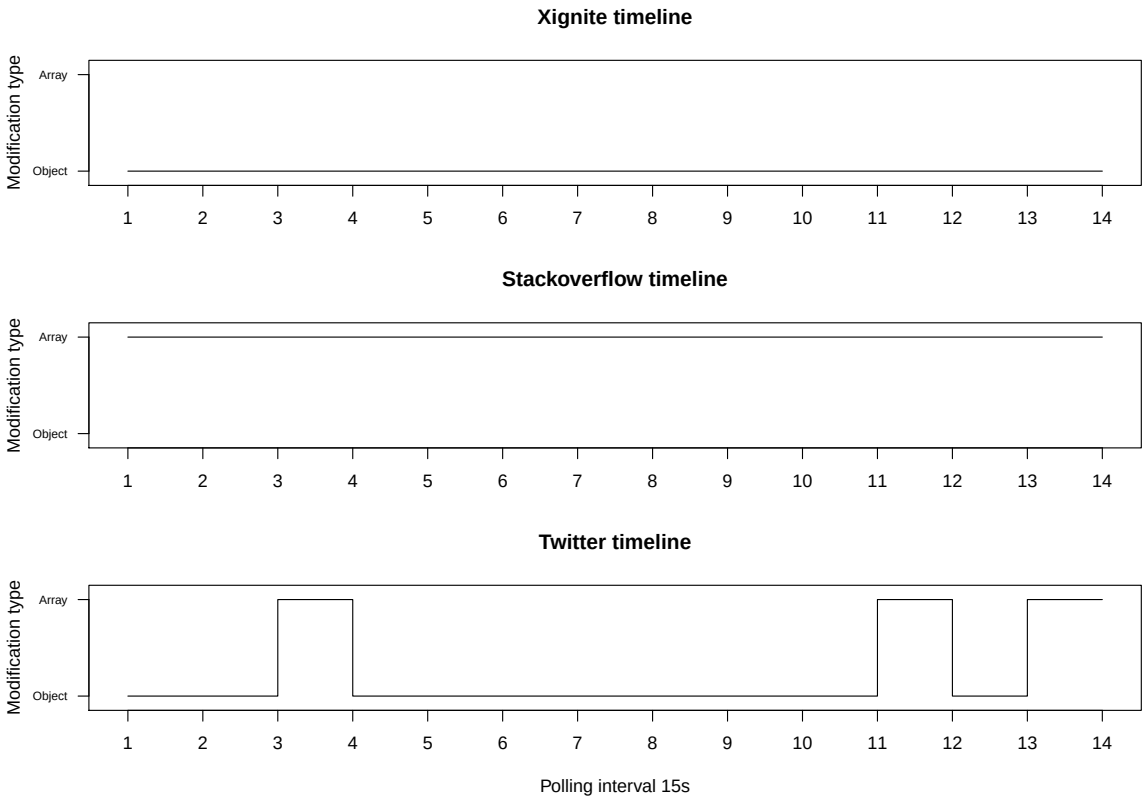


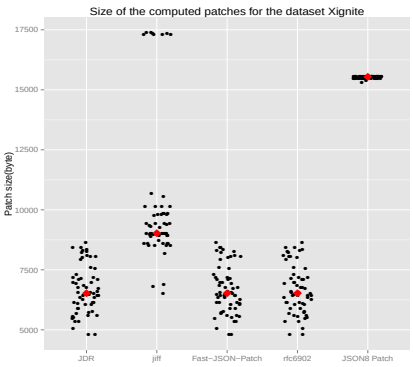
Figure 5.3 – Timeline modification type analysis for Xignite (top), Stackoverflow (middle) and Twitter (bottom), which represent *object server*, *array server* and *shift server* respectively.

library but the difference is no more than 0.5 milliseconds (see Figure 5.4b). rfc6902 takes much more time than the others.

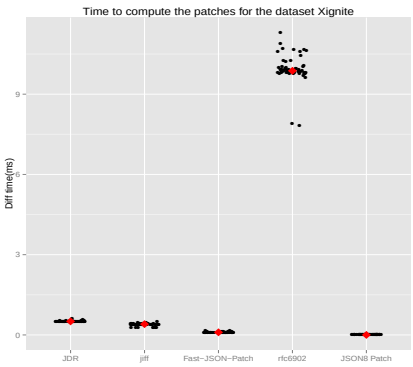
For the Stackoverflow service it is interesting to see that Fast-JSON-Patch performs bad in term of size as it generates large patches (see Figure 5.5a). JSON8 is again quite bad as it generated also large patches. JDR, jiff, rfc6902 behave quite well regarding size as they always yield small patches. Regarding time all the libraries behave quite well but rfc6902, which is slower (see Figure 5.5b).

For the Twitter service, the Figure 5.6a clearly shows that JDR always yields small patches in all situation. In some cases, Fast-JSON-Patch totally fails (see some black dots with high patch sizes). rfc6902 succeeds almost all the times but is sometimes not that fast. Regarding time, the Figure 5.6b shows that JSON8 is definitively the fastest, then Fast-JSON-Patch. JDR and Jiff performs almost within the same time. Finally rfc6902 is slow.

The Figure 5.4, 5.5, 5.6 are consistent with the analyse we provided in the section 2.3.4,



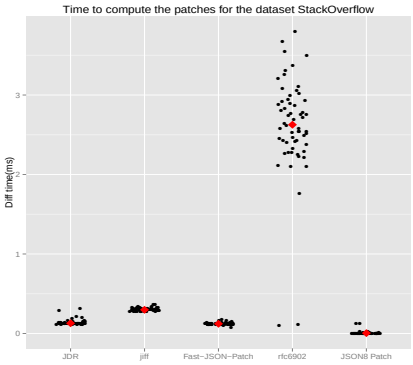
(a) Patch size



(b) Diff time

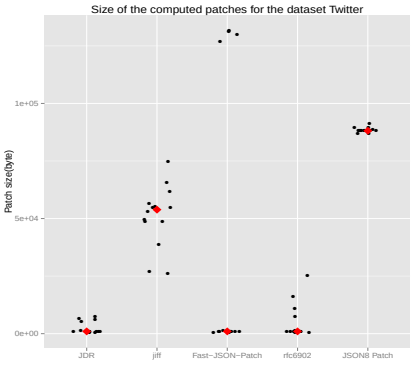
Figure 5.4 – Results for the Xignite dataset

(a) Patch size

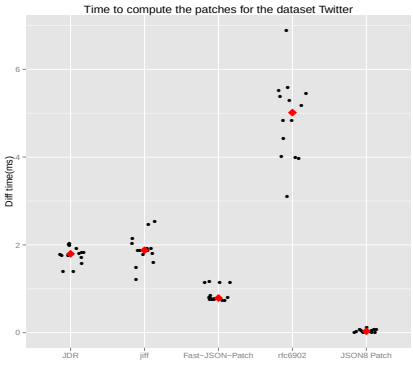


(b) Diff time

Figure 5.5 – Results for the StackOverflow dataset



(a) Patch size



(b) Diff time

Figure 5.6 – Results for the Twitter dataset

Table 5.1 – Xignite performance of the 5 existing JavaScript libraries.

	Xignite		
Library	Patch-Size	Diff-time	**Total-time**
Fast-JSON-Patch	100% (6683Bytes)	100% (0.103ms)	100% (5.45ms)
JDR	100%	502%	108%
jiff	152%	385%	157%
JSON8 Patch	232%	2%	228%
rfc6902	100%	2531%	281%

Table 5.2 – Stackoverflow performance of the 5 existing JavaScript libraries.

	Stackoverflow		
Library	Patch-Size	Diff-time	**Total-time**
JDR	100% (2257Bytes)	100% (0.123ms)	100% (1.94ms)
jiff	104%	216%	112%
rfc6902	100.3%	1880%	228%
JSON8 Patch	232%	6%	484%
Fast-JSON-Patch	1045%	88%	995%

and clearly show the advantages and drawbacks of the existing libraries, depending on the support they provide to object or array. We then decided to combine the size and time factors considering that a patch has to be sent into the internet after it has been created (with a bandwidth of 10 Mbit/s) (See Tables 5.1, 5.2 and 5.3). The Table 5.1 shows that Fast-JSON-Patch is the best when the changes are only performed to the objects' properties but our library JDR is very close. Then, the Table 5.2 shows that our library JDR performs the best when the changes are only performed to arrays. Finally, the Table 5.3 shows that our library JDR performs the best when the changes are performed to both objects' properties and arrays.

In conclusion, based on industrial real data, our JDR outperforms existing libraries regarding the size of created patches and the time needed to create them.

5.4 Conclusion

REST APIs together with JSON are commonly used by modern web applications to export their services. Such an architecture however makes the services reachable in a pull mode which is not suitable for accessing data that periodically changes (such as Twitter timeline, realtime currency, etc.). The push mode is on the contrary more adequate for

Table 5.3 – Twitter performance of the 5 existing JavaScript libraries.

	Twitter		
Library	Patch-Size	Diff-time	**Total-time**
JDR	100% (2475Bytes)	100% (1.77ms)	100% (3.75ms)
rfc6902	198%	276%	235%
Fast-JSON-Patch	1525%	50%	827%
jiff	2064%	107%	1140%
JSON8 Patch	3575%	3%	1887%

accessing changing data, but very few web applications, if any, support it. Our partner StreamData.IO therefore provides a proxy server solution for turning a pull REST API into a push one. The proxy server makes periodical requests to the API and then generates patches that express the changes made to the new received versions of the data. Generating a patch for JSON document is obviously the difficult part and existing approaches handle it poorly. In this chapter we then provide a new JSON patch algorithm towards this issue, with the objective to fully support the JSON patch RFC and to provide efficiency gain in comparison to existing libraries.

Our study first shows that the existing approaches are not optimal and that they take drastic simplifications. More precisely, we show that existing approaches do not support the *move* and *copy* change operations (except Java JSON-patch), and that few of them fully support changes performed to the array.

We then propose our JSON patch algorithm that is compliant with the JSON Patch RFC and that further supports all of the five change operations. Indeed, our algorithm succeeds to support *move* and *copy* operations for object nodes and arrays. Its limitation is that it only considers changes that are performed on a whole sub-tree, and does not consider changes that modify the structure of a sub-tree. Further, it only considers the change to the array that is localized in the same place in the two versions of a JSON document. Those limitations have however been driven by our observations performed on existing REST API, which showed that such changes almost never happen. Our approach only handles the transformation of pull mode services into push mode but not their updates. The future work is to study the subsequent API updates that may involve structural changes, which aims to better understand how far APIs are updated.

We evaluate the efficiency of the JavaScript implementation of our algorithm against existing JavaScript libraries that support the JSON Patch RFC. The evaluation has been done by requesting real web applications with data suggested by our industrial partner. It clearly shows that our library outperforms the other libraries. It creates a small patch quite fast, and can handle different situations (where the changes target objects' properties or arrays).

As a main conclusion, we provide an efficient algorithm to create a path between two versions of a JSON document. The patch created by our approach fully complies with RFC. Even it is not optimal, it however expresses all change operations such as the *move* and *copy* ones, and the ones that target arrays. Our work is the essential part for turning a pull REST API into a push one, which is frequently requested by the web developers to get notified of data changes. As an example, we provide a prototype framework that can be used to convert a pull service into a push one (see the online demo [8]).

8. http://diff-and-patch.pubstorm.site/

CHAPTER

6 Conclusion

6.1 Summary of contributions

Web applications are highly popular and using some of them (e.g., Facebook, Google) is becoming part of our lives. Developers are eager to create various web applications to meet people's increasing demands. To build a web application, developers need to know some basic programming technologies. Moreover, they prefer to use some third-party components (such as server-side libraries, client-side libraries, REST services) in the web applications. By including those components, they could benefit from maintainability, reusability, readability, and efficiency. Thanks to some third-party components, they could also retrieve external data that is essential to their business logic.

In this thesis, we propose to help developers to use third-party components in their web applications. Since there are several third-party components and each component has its characteristic, we zoom into those general questions and analyze the existing works (see Table 1.1). We then decide to provide concrete solutions for these three specific sub-problems:

— SQ1: What are the best JavaScript libraries to use?

— SQ2: How to get the standard specifications of REST services?

— SQ3: How to adapt to the data changes of REST services?

Our first contribution, presented in Chapter 3, aims at identifying and recommending the JavaScript libraries to the web application developers. We provide an approach ARJL [1] that uses both syntactical and dynamical analysis of the online resources of the web applications and detects their used JS libraries. ARJL combines three different strategies: (1) search for names of libraries in the header comment of the JS files, (2)check if a JS file used

1. https://github.com/kenmick/WebCrawler

by the web application is similar to a file that is known to be a JS library, and (3) insert at runtime a sensor in the web application with the objective to dynamically detection. By applying ARJL on the 100 most popular websites referenced by Alexa, we provide a popularity rank of JavaScript libraries used in these websites. Developers could seek suggestions when they have trouble to choose a library to include within their own web application. Moreover, thanks to our ARJL, we are able to provide version-level recommendations for popular libraries (e.g., jQuery). As a last result, we make observations about how the JavaScript library evolves over three years. The result shows that the library usage evolves but not that fast. This contribution has been published in the *Proceedings of the Symposium on Applied Computing (2017)* [Cao et al., 2017b].

Our second contribution, presented in Chapter 4, aims at automatically transforming an HTML documentation into an OpenAPI specification. Employing the specification is the best practice to get knowledge of a third-party component. Since the specification is standard and can be read by machine for automation (e.g., automated requesting, automated testing). Our ExtrateREST [2] approach inputs a single index URL, gets all the documentation pages that are linked by it, selects the useful ones thanks to a machine-learning algorithm, extracts the endpoint information according to a manually-built extraction configuration, and then produces the corresponding OpenAPI specification. The validation we performed shows that ExtrateREST produces good results for popular REST services as well as randomly selected ones. We also provide a public directory of OpenAPI specifications for REST services. Our contribution presents a way to get the standard specification from existing documentation. Even though our approach is designed for REST service, it can be applied to other third-party components. Since third-party components usually provide API documentations and the extraction configurations can be defined to excerpt the target information. This contribution has been published in the *International Conference on Service-Oriented Computing (2017)* [Cao et al., 2017a].

Our third contribution, presented in Chapter 5, aims at adapting to the data changes of REST services. Existing REST services usually support pull mode request which web applications need to call the service periodically for accessing changing data. The push mode is on the contrary more adequate for accessing changing data, but very few web applications, if any, support it. Our partner StreamData.IO therefore provides a proxy server solution for turning a pull REST API into a push one. The proxy server makes periodical requests to the API and then generates patches that express the changes made to the new received versions of the data. Generating a patch for JSON document is obviously the difficult part and existing approaches handle it poorly. We present a new JSON patch algorithm, named JDR [3], that drastically improves the efficiency of the pull to push conversion. We use real industrial data to evaluate our algorithm against existing JavaScript libraries that support the JSON Patch RFC. It clearly shows that our library outperforms the other libraries. It gen-

2. https://github.com/caohanyang/ExtrateREST
3. https://github.com/caohanyang/json_diff_rfc6902

erates smallest patch in most cases and quite fast. As a last result, we provide a prototype framework that can be used to convert a pull service into a push one. This contribution has been published in the *International Conference on Web Engineering (2016)* [Cao et al., 2016].

6.2 Perspectives

As demonstrated throughout this thesis, helping developers to use third-party components in the web applications is a general research domain. This leaves room for some interesting research investigations. We propose here several perspectives that allow to extend the works presented in this thesis.

6.2.1 What are the best JavaScript libraries to use?

In Sec 3.5, we showed some statistics on how famous web applications are using libraries. We then give some suggestions to developers based on the library usage of the most famous web applications. One of the threat to validity is our popularity indicator just involve *present*. Precisely, it only reveals the library usage of the famous web applications at present. However, we could add more indicators to the predict the *future*, such as the most promising library, the worst breaking bad. A web application developer, the best choice for him is a library that is popular right now and would not be out of fashion in a few periods. Teyton et al. [Teyton et al., 2014] presents a migration graph that composes a synthetical indicator to rank the Java library. Inspired by this idea, we could recommend a JavaScript library according to a synthetical indicator. The synthetical indicator calculates the influence of both the *present* and *future*.

To recommend a JavaScript library to a client, we choose to use user-independent property (i.e., popularity) to compose the ranking. According to Table 2.1, component recommendation is a hot research topic, and several works have been done on other third-party components. They employ collaborative filtering [Rich, 1979] approach to mine the similarity among user-dependent properties. We need to build a dataset that contains users' properties and the JavaScript libraries they use. Based on the dataset, we can recommend a JS library to a potential user.

6.2.2 How to get the standard specifications of REST services?

In Sec 4.4, we showed the quantitative performance evaluation we measure the four mandatory parts of a REST specification: Base URL, Path Templates, Verbs, and Parameters. The future work would be improve the performance and extract more parts to complete a REST specification.

— We plan to use machine learning algorithms to improve the precision/recall results for the Base URL and Path Templates parts. Our existing approach used Random forest classification algorithm to select the useful HTML pages. It would be better if we can use classification algorithms to filter useful URLs, as Yang et al. [Yang et al., 2018; Dolby et al., 2018] did. Since HTML pages contain several URLs that are not related to the API. It is hard to set a single rule to remove these noises.

— Despite of the four mandatory parts of a REST specification, there are also some parts that are essential to the developers, such as Response example, Authentication method. We plan to extract those components from HTML documentation, to generate a complete REST specification.

— We are exploring how to use real API calls to enhance the specification produced by ExtrateREST.

6.2.3 How to adapt to the data changes of REST services?

In Chapter 5, we showed the solution of how to adapt the data changes of REST services. While web applications also need to adapt to the new version of the third-party component, which is related to version migration. In the future, we want to target the version migration problem for REST services. As shown in Table 2.1, many researchers show their interests in this topic. One of the challenges is to analyze historical API documentations for massive REST services. The existing work manually check historical API documentations and thus they get some statistic for just a few REST services. In the future, we plan to provide an automatic approach to monitor massive REST services and give migration suggestions to developers. The structure of the approach would be:

— Use ExtrateREST to crawl the historical HTML documentations of REST services, and generate specifications of each version.

— Use our JSON Patch algorithm JDR to compare the two versions of REST API specifications and generate the patch. Since REST API specification is actually a JSON document.

— The generated patch shows the difference of two versions and exactly presents how the REST service evolves.

The approach is fully automated and can be applied to a large set of REST services. We would use this approach to reveal the trend of REST service evolution, or alert developers when their registered REST services have change.

APPENDIX

A Résumé en Français

Les applications Web sont très populaires et l'utilisation de certaines d'entre elles (p. ex. Facebook, Google) fait de plus en plus partie de nos vies. Les développeurs sont impatients de créer diverses applications Web pour répondre à la demande croissante. Pour construire une application web, les développeurs doivent connaître certaines technologies de programmation fondamentales, telles que HTML, JavaScript, CSS, système de base de données. De plus, ils préfèrent utiliser des composants tiers dans les applications Web. Nous appelons composants tiers un ensemble de modules ou de fonctionnalités réutilisables qui sont fournis par des fournisseurs tiers. Ils ont été largement utilisés pour rendre le développement plus facile, moins cher et de meilleure qualité. Les composants tiers comprennent les bibliothèques, les services REST et d'autres types de modules. Dans cette thèse, nous nous concentrons sur les bibliothèques et les services REST.

— Une bibliothèque est un morceau de code réutilisable qui aide les développeurs à gérer les détails de bas niveau et à réaliser leur logique métier. Selon l'endroit où elle se trouve, elle peut être classée comme bibliothèque côté serveur ou côté client. La bibliothèque côté serveur peut être utilisée pour connecter la base de données, gérer la communication HTTP, automatiser les tests, récupérer les données du service REST, etc. Par exemple, *ExpressJS* est une bibliothèque côté serveur pour définir l'application, configurer le routeur et gérer les requêtes. La bibliothèque côté client est généralement écrite en JavaScript et peut être exécutée par le navigateur. Il peut s'agir d'un ensemble de fonctionnalités pour aider les développeurs à gérer les éléments HTML DOM, cookies, requêtes HTTP, etc. Par exemple, *jQuery* facilite la manipulation d'un document HTML, la sélection d'éléments DOM, la création d'animations et la gestion des événements.

— Un service REST donne accès à un ensemble de ressources [Fielding and Taylor, 2002]. Suivant les principes REST, les accès aux ressources se font grâce à des re-

quêtes HTTP, où le verbe utilisé par la requête définit comment la ressource est manipulée (`GET` pour lire, `PUT` pour écrire, etc.). Par exemple, Instagram fournit un service REST qui donne accès aux ressources médias publiées par ses utilisateurs (photos, films, etc.).

En incluant les composants tiers, les développeurs peuvent bénéficier de la maintenabilité, de la réutilisabilité, de la lisibilité et de l'efficacité. Grâce à certains composants tiers, ils peuvent également récupérer des données externes essentielles à leur logique métier.

Dans cette thèse, nous nous concentrons sur l'aide aux développeurs pour l'utilisation de composants tiers dans les applications Web. Pour un développeur d'application web, la première question est *RQ1: quels sont les meilleurs composants tiers à utiliser?* La recommandation des composants de développement à un utilisateur potentiel a été largement étudiée par les chercheurs. Elle comporte principalement deux étapes: 1) obtenir l'ensemble de données sur l'utilisation des bibliothèques par des tiers, et 2) trier les bibliothèques en fonction de diverses exigences sous-jacentes (p. ex. popularité des composants, similarité des composants). Plusieurs travaux ont été réalisés pour obtenir l'utilisation de bibliothèques ou de services REST à partir de projets open source (voir détails dans la section 2.1). Nous avons choisi de construire l'ensemble de données d'utilisation de la bibliothèque JavaScript à partir d'applications web célèbres. Puisqu'il s'agit souvent de projets fermés et donc difficiles à analyser. Par conséquent, nous voulons répondre à la question *SQ1: Quelles sont les meilleures bibliothèques JavaScript à utiliser?*

Après avoir choisi les composants, il aurait la deuxième question: *RQ2: comment connaître les composants tiers ?* Les développeurs d'applications Web doivent connaître les composants tiers pour pouvoir les utiliser. La plupart des composants tiers fournissent généralement des documentations en ligne de leurs produits. Toutefois, ces documentations en ligne ne suivent pas la même structure ou spécification, ce qui n'est pas approprié pour l'automatisation. Les développeurs d'applications Web sont impatients d'obtenir les spécifications standard des composants tiers, qui peuvent les aider à accélérer leur développement. Cependant, il existe rarement de telles spécifications pour des composants tiers. Dans notre thèse, nous visons à répondre à cette question *SQ2: Comment obtenir les spécifications standard des services REST?* Quatre approches automatisées ou semi-automatiques: SpyREST [Sohan et al., 2017, 2015], APIDiscoverer [Ed-douibi et al., 2017], RESTler [Alarcón and Wilde, 2010], et D2Spec [Yang et al., 2018; Dolby et al., 2018], sont liés à la création des spécifications OpenAPI. Les approches basées sur des exemples SpyREST et APIDiscoverer s'appuient sur les informations d'appels API données manuellement (par exemple, URL, HTTP Verb) dans la documentation HTML correspondante. Afin d'obtenir l'ensemble du spectre, les développeurs doivent également trouver récursivement les appels de l'API pour tous les points finaux. Ce travail prend beaucoup de temps et de travail. Les approches basées sur des crawlers RESTler et D2Spec sont alimentées par un index URL de la documentation en ligne. En parcourant automatiquement la doc-

umentation HTML, les développeurs n'ont pas besoin de saisir manuellement tous les appels API. Cependant, les spécifications obtenues ne sont pas complètes, comparées dans le tableau 2.2.

Le développeur sait comment utiliser les composants tiers et les intègre avec succès dans l'application Web. Au fil du temps, les composants tiers peuvent être modifiés. L'application web doit alors s'adapter en conséquence. Voici donc la troisième question: *RQ3: comment s'adapter aux changements de composants tiers?* Nous nous concentrons sur l'adaptation des données, ce qui signifie que les applications Web doivent récupérer efficacement les données fréquemment mises à jour. L'adaptation des données n'existe que dans le cadre du service REST puisque les autres composants ne renvoient pas de données. Par conséquent, notre objectif est de répondre à cette question *SQ3: Comment s'adapter aux changements de données des services REST?* La méthode actuelle de communication avec le service REST consiste à appeler le service périodiquement, ce qui n'est pas efficace lorsque les données changent fréquemment et de façon imprévisible. La plupart des fournisseurs de services REST ont conçu leurs services pour être utilisés en mode pull selon notre enquête. Les circonstances actuelles dérangent les développeurs web côté consommateurs puisqu'ils préfèrent les services en mode push, mais ils n'ont pas accès pour changer l'infrastructure de ces services. Par conséquent, les développeurs d'applications Web sont impatients de disposer d'une plate-forme qui transforme un service en mode pull en un service en mode push sans modifier les codes sources. Certaines entreprises commerciales comme StreamData.io [1], Diffusion [2] ont déjà trouvé et soutiennent le besoin. Même s'ils suivent le même principe de conversion, l'algorithme fondamental qu'ils utilisent n'est pas parfait. Les algorithmes de patchs JSON existants ne génèrent pas de patchs optimaux et ne tirent pas profit de certaines opérations de patchs définies dans le RFC de patchs JSON.

En résumé, nous nous concentrons sur les trois points suivants:

— SQ1: Quelles sont les meilleures bibliothèques JavaScript à utiliser?

— SQ2: Comment obtenir les spécifications standard des services REST?

— SQ3: Comment s'adapter aux changements de données des services REST?

Notre première contribution, présentée dans Chapter 3, vise à identifier et recommander les bibliothèques JavaScript aux développeurs d'applications web. Nous proposons une approche ARJL qui utilise une analyse syntaxique et dynamique des ressources en ligne des applications web et détecte leurs bibliothèques JS utilisées. ARJL combine trois stratégies différentes: (1) rechercher des noms de bibliothèques dans le commentaire d'entête des fichiers JS, (2) vérifier si un fichier JS utilisé par l'application Web est similaire à un fichier connu pour être une bibliothèque JS, et (3) insérer au moment de l'exécution

1. http://streamdata.io/
2. https://www.pushtechnology.com/

un capteur dans l'application Web avec pour objectif une détection dynamique. En appliquant ARJL sur les 100 sites Web les plus populaires référencés par Alexa, nous fournissons un classement de popularité des bibliothèques JavaScript utilisées dans ces sites. Les développeurs pourraient solliciter des suggestions lorsqu'ils ont des difficultés pour choisir une bibliothèque à inclure dans leur application Web. De plus, grâce à notre ARJL, nous sommes en mesure de fournir des recommandations au niveau des versions pour les bibliothèques populaires (par exemple, jQuery). Enfin, nous faisons des observations sur l'évolution de la bibliothèque JavaScript sur trois ans. Le résultat montre que l'utilisation de la bibliothèque évolue mais pas si vite.

Notre deuxième contribution, présentée dans Chapter 4, vise à transformer automatiquement une documentation HTML en une spécification OpenAPI. L'utilisation de la spécification est la meilleure pratique pour prendre connaissance d'un artefact d'une tierce partie. Puisque la spécification est standard et peut être lue par une machine pour l'automatisation (p. ex. demande automatisée, test automatisé). Notre approche ExtrateREST saisit une URL d'index unique, obtient toutes les pages de documentation qui y sont liées, sélectionne celles qui sont utiles grâce à un algorithme d'apprentissage machine, extrait les informations du point final selon une configuration d'extraction construite manuellement, et produit ensuite la spécification OpenAPI correspondante. La validation que nous avons effectuée montre qu'ExtrateREST produit de bons résultats pour les services REST populaires ainsi que pour les services sélectionnés au hasard. Nous fournissons également un répertoire public des spécifications OpenAPI pour les services REST. Notre contribution présente un moyen d'obtenir la spécification standard à partir de la documentation existante. Bien que notre approche soit conçue pour le service REST, elle peut être appliquée à d'autres composants tiers. Étant donné que les composants tiers fournissent généralement des documentations API et que les configurations d'extraction peuvent être définies pour extraire les informations cibles.

Notre troisième contribution, présentée au chapitre 5, vise à s'adapter aux changements de données des services REST. Comme nous l'avons vu plus haut, les développeurs d'applications Web sont impatients de disposer d'une plate-forme qui transforme un service en mode pull en un service en mode push. La plate-forme effectue des requêtes périodiques à l'API et génère ensuite des correctifs qui expriment les modifications apportées aux nouvelles versions reçues des données. Générer un correctif pour un document JSON est évidemment la partie difficile et les approches existantes le gèrent mal. Nous présentons un nouvel algorithme de patch JSON, appelé JDR, qui améliore considérablement l'efficacité de la conversion pull vers push. Nous utilisons des données industrielles réelles pour évaluer notre algorithme par rapport aux bibliothèques JavaScript existantes qui prennent en charge le patch JSON RFC. Cela montre clairement que notre bibliothèque est plus performante que les autres bibliothèques. Il génère le plus petit patch dans la plupart des cas et assez rapidement. Enfin, nous fournissons un prototype de cadre qui peut être utilisé pour convertir un service pull en service push.

Enfin, le chapitre 6 conclut cette thèse en résumant nos contributions, et en présentant plusieurs perspectives possibles pour élargir notre travail.

Bibliography

Agrawal, R., Srikant, R., et al. (1994). Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499. Cited page 14.

Al-Ekram, R., Adma, A., and Baysal, O. (2005). diffX: An Algorithm to detect Changes in Multi Version XML Documents. In *In processing of the CASCON '05*, pages 1–11. Cited page 31.

Alarcón, R. and Wilde, E. (2010). Restler: crawling restful services. In *Proceedings of the 19th international conference on World wide web*, pages 1051–1052. ACM. Cited pages 21 and 88.

Baldassarre, M. T., Bianchi, A., Caivano, D., and Visaggio, G. (2005). An Industrial Case Study on Reuse Oriented Development. In *Proceedings of the 21st IEEE ICSM*, ICSM '05, pages 283–292, Washington, DC, USA. IEEE Computer Society. Cited pages 2 and 6.

Baskerville, R., Ramesh, B., Levine, L., Pries-Heje, J., and Slaughter, S. (2003). Is Internet-Speed Software Development Different? *IEEE Software*, 20(6):70–77. Cited page 6.

Bauer, V., Heinemann, L., and Deissenboeck, F. (2012). A structured approach to assess third-party library usage. In *28th IEEE ICSM 2012, Trento, Italy, September 23-28, 2012*, pages 483–492. Cited page 34.

Beverloo, P., Sullivan, B., Fullea, E., Thomson, M., and van Ouwerkerk, M. (2017). Push API. W3C working draft, W3C. https://www.w3.org/TR/2017/WD-push-api-20171215/. Cited page 25.

Bibeault, B. and Katz, Y. (2008). *Jquery in Action*. Manning Publications Co., Greenwich, CT, USA. Cited page 34.

Bille, P. (2005). A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239. Cited page 30.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022. Cited page 14.

Bryan, P. and Nottingham, M. (2013). JavaScript Object Notation (JSON) Patch. Technical report, RFC 6902, April. Cited pages 10, 29, and 66.

Buttler, D. (2004). A short survey of document structure similarity algorithms. In *International conference on internet computing*, pages 3–9. Cited pages 8 and 66.

Cao, H., Falleri, J.-R., and Blanc, X. (2017a). Automated generation of rest api specification from plain html documentation. In *International Conference on Service-Oriented Computing*, pages 453–461. Springer. Cited pages 51, 54, 55, and 84.

Cao, H., Falleri, J.-R., Blanc, X., and Zhang, L. (2016). Json patch for turning a pull rest api into a push. In *International Conference on Service-Oriented Computing*, pages 435–449. Springer. Cited page 85.

Cao, H., Peng, Y., Jiang, J., Falleri, J.-R., and Blanc, X. (2017b). Automatic identification of client-side javascript libraries in web applications. In *Proceedings of the Symposium on Applied Computing*, pages 670–677. ACM. Cited page 84.

Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. (1996). Change Detection in Hierarchically Structured Information. In Jagadish, H. V. and Mumick, I. S., editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 493–504. ACM Press. Cited page 30.

Chen, C., Gao, S., and Xing, Z. (2016). Mining analogical libraries in q&a discussions–incorporating relational and categorical knowledge into word embedding. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 338–348. IEEE. Cited pages 12, 13, and 14.

Chen, C. and Xing, Z. (2016). Similartech: automatically recommend analogical libraries across different programming languages. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 834–839. IEEE. Cited pages 13 and 14.

Chinnici, R., Moreau, J.-J., Ryman, A., and Weerawarana, S. (2007). Web services description language (wsdl) version 2.0 part 1: Core language. *W3C recommendation*, 26:19. Cited pages 7 and 50.

Cobena, G., Abiteboul, S., and Marian, A. (2002). Detecting Changes in XML Documents. In Agrawal, R. and Dittrich, K. R., editors, *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 41–52. IEEE Computer Society. Cited page 31.

Conallen, J. (1999). Modeling web application architectures with uml. *Communications of the ACM*, 42(10):63–70. Cited page 2.

Cooley, R., Mobasher, B., and Srivastava, J. (1997). Web mining: Information and pattern discovery on the world wide web. In *Tools with Artificial Intelligence, 1997. Proceedings., Ninth IEEE International Conference on*, pages 558–567. IEEE. Cited page 50.

Crockford, D. (2006). *RFC4627: JavaScript Object Notation*. Cited page 66.

Danielsen, P. J. and Jeffrey, A. (2013). Validation and interactivity of web api documentation. In *Web Services (ICWS), 2013 IEEE 20th International Conference on*, pages 523–530. IEEE. Cited pages 7, 17, and 50.

Davidson, J. D. and Coward, D. (1999). Java servlet specification ("specification") version: 2.2 final release. Technical report. Cited page 2.

Dice, L. R. (1945). Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302. Cited page 37.

Dig, D., Negara, S., Mohindra, V., and Johnson, R. (2008). Reba: re factoring-aware b inary a daptation of evolving libraries. In *Proceedings of the 30th international conference on Software engineering*, pages 441–450. ACM. Cited page 5.

Dolby, J. T., Wittern, J. E., Yang, J., and Ying, A. T. (2018). Generating web api specification from online documentation. US Patent App. 15/403,150. Cited pages 21, 86, and 88.

Ed-douibi, H., Izquierdo, J. L. C., and Cabot, J. (2017). Example-driven web api specification discovery. In *European Conference on Modelling Foundations and Applications*, pages 267–284. Springer. Cited pages 16, 21, and 88.

Ellis, C. A. and Gibbs, S. J. (1989). Concurrency control in groupware systems. In *Acm Sigmod Record*, volume 18, pages 399–407. ACM. Cited page 71.

Espinha, T., Zaidman, A., and Gross, H.-G. (2014). Web api growing pains: Stories from client developers and their code. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 84–93. IEEE. Cited page 5.

Fielding, R. T. and Taylor, R. N. (2002). Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150. Cited pages 4, 15, 50, 66, and 87.

Fokaefs, M. and Stroulia, E. (2015). Using wadl specifications to develop and maintain rest client applications. In *Web Services (ICWS), 2015 IEEE International Conference on*, pages 81–88. IEEE. Cited pages 7 and 50.

Gellersen, H.-W. and Gaedke, M. (1999). Object-oriented web application development. *IEEE Internet Computing*, 3(1):60–68. Cited page 2.

Hadley, M. (2009). Web application description language: W3c member submission 31 august 2009. Technical report, World Wide Web Consortium. Cited page 16.

Hamming, R. W. (1950). Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160. Cited page 37.

Hersak, D. (2017). Donald trump tweet statistics | kaggle. Cited page 24.

Higuchi, S., Kan, T., Yamamoto, Y., and Hirata, K. (2012). An A* algorithm for computing edit distance between rooted labeled unordered trees. In *New Frontiers in Artificial Intelligence*, pages 186–196. Springer. Cited page 30.

Hirschberg, D. S. (1977). Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675. Cited page 71.

Ho, T. K. (1995). Random decision forests. In *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, volume 1, pages 278–282. IEEE. Cited page 54.

Hogue, A. and Karger, D. (2005). Thresher: automating the unwrapping of semantic content from the world wide web. In *Proceedings of the 14th international conference on World Wide Web*, pages 86–95. ACM. Cited pages 51 and 55.

Ishio, T., Kula, R. G., Kanda, T., German, D. M., and Inoue, K. (2016). Software ingredients: Detection of third-party component reuse in java software release. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 339–350. ACM. Cited pages 12 and 13.

Jiménez, P. and Corchuelo, R. (2016). On learning web information extraction rules with tango. *Information Systems*, 62:74–103. Cited pages 51 and 55.

Katsuragawa, D., Ihara, A., Kula, R. G., and Matsumoto, K. (2018). Maintaining third-party libraries through domain-specific category recommendations. In *2018 IEEE/ACM 1st International Workshop on Software Health (SoHeal)*, pages 2–9. IEEE. Cited pages 12, 13, and 14.

Klyne, G. and Carroll, J. (2004). Resource description framework (RDF): Concepts and abstract syntax. W3C recommendation, W3C. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. Cited page 17.

Kopeckỳ, J., Gomadam, K., and Vitvar, T. (2008). hrests: An html microformat for describing restful web services. In *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT'08. IEEE/WIC/ACM International Conference on*, volume 1, pages 619–625. IEEE. Cited pages 16 and 17.

Koprinska, I., Poon, J., Clark, J., and Chan, J. (2007). Learning to classify e-mail. *Information Sciences*, 177(10):2167–2187. Cited page 54.

Kula, R., German, D., Ishio, T., and Inoue, K. (2015). Trusting a library: A study of the latency to adopt the latest Maven release. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 520–524. Cited page 34.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565. Cited page 71.

Lee, B. (2012). A temporal analysis of posting behavior in social media streams. In *International AAAI conference on weblogs and social media*. Cited pages 24 and 25.

Li, J., Xiong, Y., Liu, X., and Zhang, L. (2013). How does web service api evolution affect clients? In *2013 IEEE 20th International Conference on Web Services*, pages 300–307. IEEE. Cited page 5.

Lindholm, T., Kangasharju, J., and Tarkoma, S. (2006). Fast and Simple XML Tree Differencing by Sequence Alignment. In *Proceedings of the 2006 ACM Symposium on Document Engineering*, DocEng '06, pages 75–84, New York, NY, USA. ACM. Cited page 31.

Liu, B. (2007). *Web data mining: exploring hyperlinks, contents, and usage data*. Springer Science & Business Media. Cited page 50.

López, M., Ferreiro, H., Francisco, M. A., and Castro, L. M. (2013). Automatic generation of test models for web services using wsdl and ocl. In *International Conference on Service-Oriented Computing*, pages 483–490. Springer. Cited pages 7 and 50.

Mardan, A. (2014). *Practical Node.Js: Building Real-World Scalable Web Apps*. Apress, Berkely, CA, USA, 1st edition. Cited page 34.

Melanson, D. (2008). iphone push notification service for devs announced. Cited page 25.

Meng, S., Wang, X., Zhang, L., and Mei, H. (2012). A history-based matching approach to identification of framework evolution. In *Proceedings of the 34th International Conference on Software Engineering*, pages 353–363. IEEE Press. Cited page 5.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119. Cited page 14.

Onan, A. (2016). Classifier and feature set ensembles for web page classification. *Journal of Information Science*, 42(2):150–165. Cited page 51.

Ouni, A., Kula, R. G., Kessentini, M., Ishio, T., German, D. M., and Inoue, K. (2017). Search-based software library recommendation using multi-objective optimization. *Information and Software Technology*, 83:55–75. Cited pages 12, 13, and 14.

Pawlik, M. and Augsten, N. (2011). RTED: A Robust Algorithm for the Tree Edit Distance. *PVLDB*, 5(4):334–345. Cited page 30.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830. Cited page 54.

Per, C. (2014). Web application definition. *TechTerms. Sharpened Productions*. Cited page 3.

pushtechnology (2018). Json delta streaming. Cited page 28.

Qi, X. and Davison, B. D. (2009). Web page classification: Features and algorithms. *ACM computing surveys (CSUR)*, 41(2):12. Cited pages 51 and 54.

Renzel, D., Schlebusch, P., and Klamma, R. (2012). Today's top "restful" services and why they are not restful. *Web Information Systems Engineering-WISE 2012*, pages 354–367. Cited pages 7 and 50.

Rich, E. (1979). User modeling via stereotypes. *Cognitive science*, 3(4):329–354. Cited page 85.

Sadowski, C. and Levin, G. (2007). Simhash: Hash-based similarity detection. Cited page 37.

Shi, S., Liu, C., Shen, Y., Yuan, C., and Huang, Y. (2015). Autorm: An effective approach for automatic web data record mining. *Knowledge-Based Systems*, 89:314–331. Cited pages 51 and 55.

Sia, K. C., Cho, J., and Cho, H.-K. (2007). Efficient monitoring algorithm for fast news alerts. *IEEE Transactions on Knowledge & Data Engineering*, (7):950–961. Cited page 24.

Sleiman, H. A. and Corchuelo, R. (2014). A class of neural-network-based transducers for web information extraction. *Neurocomputing*, 135:61–68. Cited pages 51 and 55.

Sohan, S., Anslow, C., and Maurer, F. (2015). Spyrest: Automated restful api documentation using an http proxy server (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 271–276. IEEE. Cited pages 18 and 88.

Sohan, S., Anslow, C., and Maurer, F. (2017). Automated example oriented rest api documentation at cisco. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, pages 213–222. IEEE Press. Cited pages 18, 20, and 88.

streamdata.io (2016). How streamdata.io turn any api into a streaming api. Cited page 27.

Tam, T. (2017). Mulesoft joins the openapi initiative: The end of the api spec wars. Cited page 17.

Terveen, L. and Hill, W. (2001). Beyond recommender systems: Helping people help each other. *HCI in the New Millennium*, 1(2001):487–509. Cited page 14.

Teyton, C., Falleri, J.-R., and Blanc, X. (2012). Mining Library Migration Graphs. In IEEE, editor, *19th WCRE Conference, 2012*, pages 289–298, Kingston, Ontario, Canada. Cited pages 6 and 34.

Teyton, C., Falleri, J.-R., and Blanc, X. (2013). Automatic Discovery of Function Mappings between Similar Libraries. In *20th WCRE Conference 2013*, pages 192–201. IEEE. Cited page 34.

Teyton, C., Falleri, J.-R., Palyart, M., and Blanc, X. (2014). A study of library migrations in Java. *Journal of Software: Evolution and Process*, 26(11):1030–1052. Cited pages 12, 13, 14, 34, and 85.

Thung, F., Lo, D., and Lawall, J. (2013). Automated library recommendation. In *20th WCRE Conference, 2013*, pages 182–191. Cited pages 12, 14, and 34.

Velloso, R. P. and Dorneles, C. F. (2017). Extracting records from the web using a signal processing approach. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 197–206. ACM. Cited pages 51 and 55.

Verborgh, R., Steiner, T., Van Deursen, D., De Roo, J., Van de Walle, R., and Vallés, J. G. (2013). Capturing the functionality of web services with functional descriptions. *Multimedia tools and applications*, 64(2):365–387. Cited page 17.

W.A, A. and S.M, E. (2011). Machine learning methods for e-mail classification. *International Journal of Computer Science & Information Technology (IJCSIT)*, 16. Cited page 54.

Wagner, F., Klöpper, B., Ishikawa, F., and Honiden, S. (2012). Towards robust service compositions in the context of functionally diverse services. In *Proceedings of the 21st international conference on World Wide Web*, pages 969–978. ACM. Cited pages 7 and 50.

Wang, H., Kessentini, M., and Ouni, A. (2016). Prediction of web services evolution. In *International Conference on Service-Oriented Computing*, pages 282–297. Springer. Cited page 5.

Wu, H. C., Luk, R. W. P., Wong, K. F., and Kwok, K. L. (2008). Interpreting tf-idf term weights as making relevance decisions. *ACM Trans. Inf. Syst.*, 26(3):13:1–13:37. Cited page 54.

Wu, Q., Wu, L., Liang, G., Wang, Q., Xie, T., and Mei, H. (2013). Inferring dependency constraints on parameters for web services. In *Proceedings of the 22nd international conference on World Wide Web*, pages 1421–1432. ACM. Cited pages 7 and 50.

Yang, J., Wittern, E., Ying, A. T., Dolby, J., and Tan, L. (2018). Towards extracting web api specifications from documentation. Cited pages 21, 86, and 88.

Yu, H., Xia, X., Zhao, X., and Qiu, W. (2017). Combining collaborative filtering and topic modeling for more accurate android mobile app library recommendation. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware*, page 17. ACM. Cited pages 12, 13, and 14.

Zeleny, J., Burget, R., and Zendulka, J. (2017). Box clustering segmentation: A new method for vision-based web page preprocessing. *Information Processing & Management*, 53(3):735–750. Cited pages 51 and 55.

Zhang, K. and Shasha, D. (1989). Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.*, 18(6):1245–1262. Cited pages 8 and 66.

Zhang, K., Statman, R., and Shasha, D. (1992). On the editing distance between unordered labeled trees. *Information processing letters*, 42(3):133–139. Cited page 30.

Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S. (2005). Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445. Cited page 34.

List of Figures

List of Tables