UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Rômulo Silva do Nascimento

EMPIRICAL EVALUATION OF API DEPRECATION IN JAVASCRIPT

Belo Horizonte
2021

Rômulo Silva do Nascimento

EMPIRICAL EVALUATION OF API DEPRECATION IN JAVASCRIPT

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Eduardo Magno Lages Figueiredo

Coorientador: Prof. Dr. André Cavalcante Hora

Belo Horizonte

2021

RÔMULO SILVA DO NASCIMENTO

EMPIRICAL EVALUATION OF API DEPRECATION IN JAVASCRIPT

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Eduardo Magno Lages Figueiredo

Co-Advisor: André Cavalcante Hora

Belo Horizonte
2021

© 2021, Rômulo Silva Do Nascimento.
Todos os direitos reservados

Silva do Nascimento, Rômulo

S586e Empirical evaluation of API deprecation in JavaScript [manuscrito] / Rômulo Silva do Nascimento — 2021.
57 f. il.

Orientador: Eduardo Magno Lages Figueiredo.
Coorientador: André Cavalcante Hora.
Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação
Referências: f. 56-57.

1. Computação – Teses. 2. –Teses. 3. APIs (Application Programing Interfaces) – Depreciação –Teses. 4. JavaScript (Linguagem de programação de computador) – Teses. 5. Qualidade de software – Teses I. Figueiredo, Eduardo Magno Lages. II. Hora, André Cavalcante. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. IV.Título.

CDU 519.6*82 (043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6/1510 – Insituto de Ciências Exatas da UFMG

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Empirical Evaluation of API Deprecation in JavaScript

RÔMULO SILVA DO NASCIMENTO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Eduardo Magno Lages Figueiredo - Orientador
Departamento de Ciência da Computação - UFMG

Prof. André Cavalcante Hora - Coorientador
Departamento de Ciência da Computação - UFMG

Prof. Marco Túlio de Oliveira Valente
Departamento de Ciência da Computação - UFMG

Prof. Marcelo de Almeida Maia
Faculdade de Computação - UFU

Belo Horizonte, 19 de Novembro de 2021.

RESUMO

Construir aplicações usando bibliotecas externas é uma prática comum no desenvolvimento de software. Como qualquer outro tipo de software, bibliotecas de código e suas APIs evoluem com o tempo. Para ajudar na migração entre versões e garantir a compatibilidade com versões anteriores, uma prática recomendada durante o desenvolvimento é depreciar APIs. Embora estudos tenham sido conduzidos para investigar depreciação em linguagens de programação como Java e C, não há estudos detalhados sobre depreciação de APIs em JavaScript. O objetivo desta dissertação é investigar a depreciação de APIs JavaScript. Portanto, apresentamos os resultados de três estudos com desenvolvedores e projetos populares em JavaScript. Primeiramente, relatamos os resultados de um estudo de survey com 109 desenvolvedores JavaScript. Então, analisamos 320 projetos JavaScript populares para identificar ocorrências de API depreciadas. Por fim, analisamos a evolução de APIs depreciadas em 50 pacotes populares JavaScript. Os resultados sugerem que não existe uma solução padrão para depreciar APIs JavaScript. No geral, encontramos várias soluções, incluindo mensagem de console, documentação externa, anotação JSDoc, comentário de código e elemento prefixado. Além disso, os desenvolvedores podem usar várias soluções para depreciação no mesmo projeto ou até mesmo no mesmo arquivo. Por fim, a maioria dos projetos apresenta tendências de crescimento no número de APIs depreciadas.

Palavras-chave: Depreciação de API. JavaScript. Evolução de Software.

ABSTRACT

Building an application using third-party libraries is a common practice in software development. As any other software system, code libraries and their APIs evolve over time. In order to help version migration and ensure backward compatibility, a recommended practice during development is to deprecate API. Although studies have been conducted to investigate deprecation in some programming languages, such as Java and C#, there are no detailed studies on API deprecation in the JavaScript ecosystem. The goal of this master dissertation is to investigate deprecation of JavaScript APIs. Therefore, we report the results of three studies with JavaScript developers and popular packages. We first report the results of a survey with 109 JavaScript developers. Afterwards, we mine 320 popular JavaScript projects to identify deprecated API occurrences. Finally, we analyze the evolution of API deprecation in 50 popular JavaScript packages. Results suggest that there is no standard solution to deprecate JavaScript APIs. Overall, we find several solutions, including console message, project documentation, JSDoc annotation, code comment, and prefixed element. Furthermore, developers may use multiple deprecation solutions in the same project or even in the same file. Additionally, most projects present upward trends in the number of deprecated APIs.

Keywords: API Deprecation. JavaScript. Software Evolution.

LIST OF FIGURES

Figure 1.1 – The 2020 State of the Octoverse Report results for the top languages over the years 12
Figure 2.1 – Stack Overflow answer in which the author recommends using JSDoc an notation with a console warning message to indication deprecation 19
Figure 2.2 – Web development blog article suggesting the use of JSDoc, comments on the deprecation context and time frame, and console warnings. 19
Figure 3.1 – Survey study methodology steps... 23
Figure 4.1 – Mining study methodology steps... 34
Figure 4.2 – JavaScript packages characteristics: npm dependents and GitHub statistics. 35
Figure 4.3 – API deprecation mechanism occurrences in JavaScript packages. 40
Figure 5.1 – Survey study methodology steps 43
Figure 5.2 – Analyzed JavaScript packages characteristics. 45
Figure 5.3 – Versions in packages. 48

LIST OF TABLES

Table 3.1 – Summary of survey questions to JavaScript developers. 26

Table 3.2 – Survey closed-ended questions and answers...27

Table 5.1 – List of selected packages and number of versions. 46

Table 5.2 – Mann-Kendall Trend Test results of deprecation occurrences on analyzed packages. 49

Table 5.3 – Mann-Kendall Trend Test results of deprecation occurrences on analyzed, considering individual deprecation mechanisms. 50

Table 5.4 – List of selected projects and number of versions. 50

CONTENTS

	CHAPTER 1	11
1	INTRODUCTION	11
1.1	Motivation	11
1.2	Proposed Work	13
1.3	Publications	14
1.4	Dissertation Outline	14
	CHAPTER 2	16
2	BACKGROUND AND RELATED WORK	16
2.1	API Deprecation	16
2.2	API Deprecation in JavaScript	17
2.3	Related Work	19
2.4	Comparison to Other Languages	20
2.5	Final Remarks	22
	CHAPTER 3	23
3	SURVEY WITH DEVELOPERS ON DEPRECATION PRACTICES	23
3.1	Goal and Research Questions	24
3.2	Study Design	24
3.3	Survey Results	25
3.3.1	The API Consumer Perspective	27
3.3.2	The API Provider Perspective	28
3.3.3	Developers' Further Insights	29
3.4	Threats to Validity	30
3.5	Final Remarks	31
	CHAPTER 4	33
4	MINING API DEPRECATION IN JAVASCRIPT PACKAGES	33
4.1	Goal and Research Questions	33
4.2	Study Design	34
4.3	Study Results	38
4.4	Threats to Validity	41
4.5	Final Remarks	42
	CHAPTER 5	43
5	ANALYSIS OF DEPRECATION EVOLUTION	43
5.1	Goal and Research Questions	43

5.2 Study Design 44
5.3 Results 47
5.4 Threats to Validity 51
5.5 Final Remarks 51
CHAPTER 6 53
6 FINAL CONSIDERATIONS 53
6.1 Work Overview 53
6.2 Contributions 54
6.3 Future Work 55
Bibliography 56

Chapter 1

1 Introduction

Building an application using third-party libraries is a common practice in software development. Libraries provide reusable functionality to client applications through their Application Programming Interfaces (APIs). API usage brings several advantages to a software development project (Tourwé & Mens, 2003), such as cost and resources usage reduction. As a result, developers can focus on business core requirements and software quality may increase by relying on libraries that have been widely adopted, tested and documented (Moser & Nierstrasz, 1996).

As any other software system, libraries and their APIs evolve over time (Granli et al., 2017). Thus, functions and parameters might be renamed, updated, moved, or removed. Consequently, client applications need to migrate to the latest stable versions of their dependencies (Bogart et al., 2016). To help version migration and ensure backward compatibility, a recommended practice in software development is to deprecate the API. In other words, deprecation indicates that the use of a certain API should be avoided because it will be changed, removed or discontinued in a future version (Robbes et al., 2012). Some of the most popular programming languages, such as Java and C#, provide native support mechanisms and tools to help developers explicitly deprecate their APIs (Sawant et al., 2018c). Indeed, recently, there have been many research on deprecation practices and mechanisms mostly on those languages (Robbes et al., 2012, Bogart et al., 2016, Brito et al., 2018, Sawant et al., 2018c, Li et al., 2018, Sawant et al., 2018a, Sawant et al., 2018b, Sawant et al., 2019). However, to the best of our knowledge, there are no detailed studies regarding API deprecation in the JavaScript ecosystem.

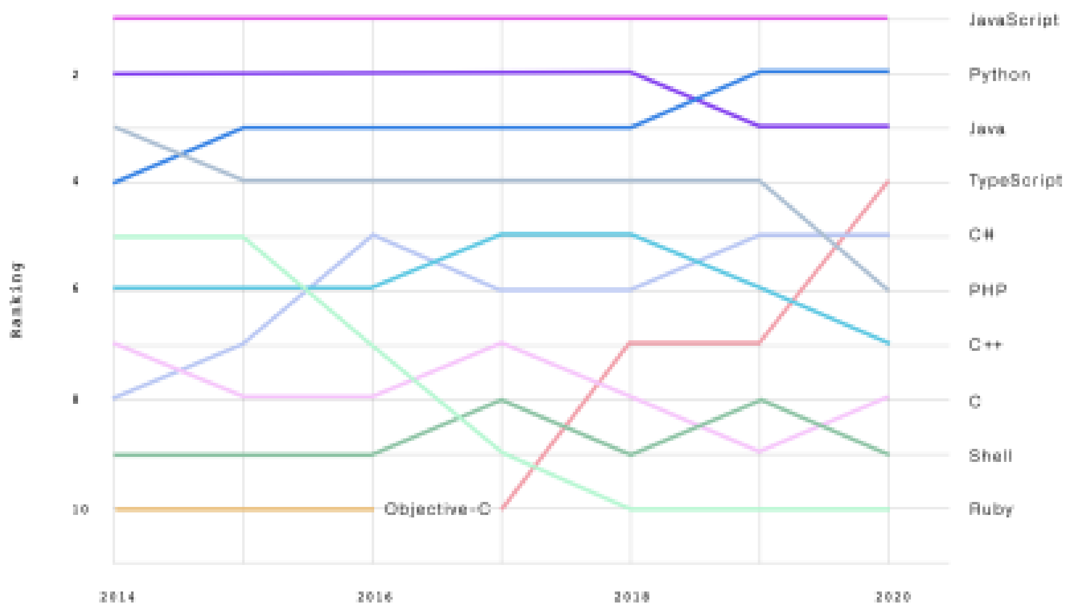
1.1 Motivation

JavaScript has become extremely popular over the last years. According to the Stack Overflow 2021 Developer Survey[1], JavaScript is the most commonly used programming language for the ninth consecutive year. As shown in Figure 1.1 chart, GitHub annual report, *The State of the Octoverse*, also indicates that JavaScript is the most popular language in terms

[1] https://insights.stackoverflow.com/survey/2021

of unique contributors to both public and private repositories[2]. The chart also shows TypeScript, a superset language of JavaScript, coming next in the fourth position. The same report also reveals that 94% of all active public repositories rely on open source software. There are over 3 million public repositories primarily written in JavaScript in GitHub. In terms of libraries, ReactJS, for instance, is a dependency of approximately 2.5 million public projects in GitHub. Despite the growth in the usage of JavaScript external libraries and APIs, little is known about JavaScript API deprecation mechanisms and practices. Additionally, there are no detailed studies related to this topic in the JavaScript ecosystem.

Figure 1.1 – The 2020 State of the Octoverse Report results for the top languages over the years.



Developers should have access to information regarding how and when deprecation in JavaScript projects is addressed. Shedding light on JavaScript deprecation practices may strongly benefit developers as they can be aware of deprecation strategies in the ecosystem. Consequently, it might lead to a clear communication environment and improve the overall quality of JavaScript projects. Furthermore, library maintainers might take advantage of this study by making informed decisions when planning for deprecation in their packages and improve communication with their clients. Ultimately, this work might identify problems and lead to actionable insights in the JavaScript ecosystem, such as creating official rules for deprecation, guidelines and conventions.

[2] https://octoverse.github.com

1.2 Proposed Work

The more APIs being leveraged by JavaScript developers, the greater consequences the deprecation of JavaScript APIs may lead to. Hence, there is a need to understand the current state of practice of JavaScript API deprecation and quantify its impacts. In order to investigate this topic, we analyze in this master dissertation the following factors:

- The API deprecation strategies adopted in JavaScript packages;
- Developers reactions to deprecated APIs in JavaScript;
- The consistency of adopted deprecation strategies within JavaScript packages;
- The evolution of deprecated APIs over time.

Our goal is to investigate the deprecation mechanisms in the JavaScript ecosystem and analyze how they are adopted and maintained. Thus, we propose the following research questions to support our study:

- RQ1: To what extent do developers see deprecated APIs and deprecate in JavaScript packages? In these questions, we investigate how to what extent consumer developers see deprecated APIs in packages they are working on, and API developers deprecate in software packages;
- RQ2: What priority do developers give to deprecation issues? In this second questions we analyze how consumer developers handle and react to deprecated APIs, if, for instance, they consider deprecation an issue that requires immediate action or not;
- RQ3: What deprecation strategies do developers most commonly see and adopt in JavaScript? In these questions we explore which deprecation strategies and mechanisms do consumer developers encounter, and API developer most commonly adopt;
- RQ4: To what extent are deprecation strategies consistent in popular JavaScript packages? In this question we analyze how deprecation mechanisms are consistently adopted within and among JavaScript packages;
- RQ5: Do deprecated APIs increase or decrease overtime in JavaScript packages? In the fourth question, we investigate how deprecated APIs evolve over time, if the amount increases, decreases or maintains stable.
- RQ6: Are deprecated APIs usually introduced and removed in major or minor releases? In this last question, we analyze when deprecated APIs are usually introduced and removed, and the observed rate of deprecated API changes in major and minor releases.

1.3 Publications

This master dissertation produced the following publications, and, therefore, it contains material of them:

- R. Nascimento, E. Figueiredo, A. Hora and A. Brito. JavaScript API Deprecation in the Wild: A First Assessment. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pages 567-571.
- R. Nascimento, E. Figueiredo and A. Hora. JavaScript API Deprecation Land scape: A Survey and Mining Study. In *IEEE Software*, 2021, vol. , no. 01, pages 0-0, 5555.
- R. Nascimento, E. Figueiredo, A. Hora and A. Brito. Exploring API Deprecation Evolution in JavaScript. In *2022 IEEE 29h International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

1.4 Dissertation Outline

The remaining portion of this Master dissertation is organized as follows:

- Chapter 2 provides background information related to this dissertation. We introduce concepts regarding Application Programming Interfaces (APIs), depreciation and briefly present the JavaScript language. Additionally, we discuss past related work and compare deprecation strategies on different popular programming languages;
- Chapter 3 describes our empirical study, a survey with developers aiming at understanding how they deal with deprecated APIs in JavaScript packages. We start by presenting the survey design and the and detail the questions that comprise it. Then, we summarize the answers we received and discuss the results and threats to validity;
- Chapter 4 presents a quantitative mining study that investigates API deprecation occurrences in popular JavaScript packages. The goal is to provide an understanding of which deprecation strategies are most commonly adopted and to what extent they are consistent among and within JavaScript packages. We first describe our mining and classification approaches. Afterwards, we discuss the study results and threats to validity;
- Chapter 5 provides a study aiming at investigating how deprecated APIs evolve over the lifetime of a JavaScript package library and analyze how the number of deprecated APIs change between version releases. We start by describing the study design, including our source code mining strategy and history analysis algorithm. Finally, we report and discuss the study results and present some threats to validity;

- Chapter 6 concludes this Master dissertation, presenting an overview of this work, main contributions, and insights for future work.

Chapter 2

2 Background and Related Work

A thorough understanding of deprecation practices and strategies adopted in the JavaScript ecosystem is relevant as insights can be used by library maintainers and consumer developers to better communicate and handle deprecated APIs. In addition, the software engineering community may define new research directions based on the comprehension of the JavaScript API deprecation landscape.

In this chapter, we present information regarding API deprecation, which is the main focus of our studies, and discuss previous related works and common strategies in other popular programming languages. Section 2.1 defines API deprecation and its role in software engineering. Section 2.2 discusses where JavaScript stands when it comes to deprecating APIs. Section 2.3 presents past studies related to API deprecation. Section 2.4 compares deprecation mechanisms in other programming languages. Finally, Section 2.5 concludes this chapter.

2.1 API Deprecation

Application Programming Interfaces (APIs) are defined as interfaces used by client software components to communicate with a software provider entity (Tourwe and Mens, 2003). JavaScript, for instance, provides standard built-in objects[3] that expose APIs to help developers with common tasks. Date.now() and Math.random() are examples of such built-in APIs. Browsers also provide their own APIs[4] to allow hosted applications to interact with some of the browser's mechanisms, such as the Console API, History API and Storage API.

Most industry level JavaScript applications rely on third-party library APIs to obtain reusable functionalities and tackle a wide variety of common problems. Listing 2.1 shows API examples of axios[5], an extremely popular JavaScript library that provides functions to make HTTP requests in a simpler way. In the example, we make a HTTP get request to a /resource endpoint. Once the request returns a response, we print it to the developer console. Listing 2.2 shows how we can execute the same task using pure JavaScript. The code is considerably longer and is not as straightforward to understand as the previous example.

[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/GlobalObjects
[4] https://developer.mozilla.org/en-US/docs/Web/API
[5] https://axios-http.com/

```
axios.get("/resource").then(function(response) {
    console.log(response);
});
```

Listing 2.1 – Example of an HTTP GET request using the axios API.

```
var req = new XMLHttpRequest();
req.open("GET", "/resource");
req.onload = function() {
  console.log(req.responseText);
}
req.send();
```

Listing 2.2 – Example of an HTTP GET request using JavaScript native code.

However, as any other type of software system, third-party libraries, such as axios, evolve over time, either by adding new functionalities or improving aspects of existing ones. Thus, methods, functions, fields, parameters and types might change. For instance, they might be renamed, moved or removed. Those sorts of API changes might imply breaking impacts on client systems (Robbes et al., 2012). In order to mitigate these impacts and help version migration, a recommended practice in software development is to deprecate APIs. In other words, deprecation indicates that the use of a certain API should be avoided because it will be changed, removed or discontinued in a future version.

2.2 API Deprecation in JavaScript

JavaScript (or JS, for short) is a versatile programming language that conforms to the ECMAScript specification[6]. It has been primarily designed and known as a language for rendering dynamic content on the client-side of Web applications. More recently, JavaScript has also been used as a server-side language through the use of the Node.js environment[7].

Software reuse has become a key factor for a cost and time efficient software development project (Uddin et al., 2011). This scenario has led to the emergence of software repositories, such as npm, that provide a centralized and simplified management and distribution of software components. The npm registry serves as a base for JavaScript Web applications, frameworks and library ecosystems. On June 4th, 2019, npm reached one million hosted JavaScript packages, making it the largest software repository to date (Tal and Maple,

[6] https://www.ecma-international.org/
[7] https://nodejs.org/en/

2019). Those libraries hosted in npm provide reusable functionality to client applications through their Application Programming Interfaces.

Unlike other popular programming languages, such as Java and C#, JavaScript provides neither native deprecation mechanisms. In adittion, there are no recommendations from ECMA International or TC39[8] on how to properly deprecate JavaScript written code. Google brings only a few relevant results when we search for "*how to deprecate JavaScript*". There is a popular answer to a Stack Overflow question. Presented in Figure 2.1, this answer recommends the use of JSDoc[9] deprecation annotation, possibly along with a console warning message indicating the deprecation[10]. In addition, Figure 2.2 shows a Web blog that also endorses the use of JSDoc – with comments on the deprecation context and time frame – and console warnings. The same blog also suggests that a deprecation utility, such as a helper function, might be suitable[11]. These recommendations are among the top results provided by a Google search. Listing 2.3 show an example of a function written in JavaScript with an indication of deprecation, using the JSDoc annotation. However, from our ad hoc searches, we observed that there is no standard way for API deprecation in JavaScript.

```
1 /**
2  * @deprecated use powerOf instead
3  */
4 function power (base) {
5   return base * base;
6 }
```

Listing 2.3 – Example of a function written in JavaScript

[8] https://www.ecma-international.org/technical-committees/tc39/
[9] https://jsdoc.app/
[10] https://stackoverflow.com/questions/19412660/how-should-i-mark-a-method-as-obsolete-in-js
[11] https://css-tricks.com/approaches-to-deprecating-code-in-javascript/

Figure 2.1 – Stack Overflow answer in which the author recommends using JSDoc an notation with a console warning message to indication deprecation.

Figure 2.2 – Web development blog article suggesting the use of JSDoc, comments on the deprecation context and time frame, and console warnings.

How to mark a method obsolete

Here are five good practices I have found the most useful:

1. Add a `@deprecated` JSDoc flag.
2. Mention the version that the method was deprecated.
3. Figure out a timeframe for when this method will be deleted, including which version it will take place. Otherwise, based on my experience, it stays forever 🙂
4. Use comments liberally to explain the implementation for the benefit of other developers or your future self. This is extremely useful if your use-case is writing a library that others use as a dependency for their work.
5. Add a console warning message that indicates that the function is deprecated.

2.3 Related Work

Previous work investigated Java API deprecation practices (Sawant et al., 2018b, Sawant et al., 2019). These works assessed the impacts, the needs, the reasons, and the patterns of API deprecation. They observed that the Java deprecation mechanism does not address all developer needs when it comes to deprecation (Sawant et al., 2018a).

Previous work also detected that Javadoc is not sufficient to understand the reasons behind deprecation occurrences. By mining other data sources, such as source code, issue tracker data, and commit history, they identified 12 reasons that trigger developers to deprecate API (Sawant et al., 2018b). They verified that most API client applications do not react to deprecation. Thus, they applied a survey to gather qualitative data from developers and try to explain this behavior (Sawant et al., 2019). Robbes at al. (2012) studied deprecation in the context of the Smalltalk ecosystem. Brito et al. (Brito et al., 2018) investigated the use of deprecation messages in Java and C#. These studies describe that 66.7% and 77.8% of Java and C# API, respectively, are deprecated with deprecation messages and that this rate does not evolve over time. Li et al. (2018) performed an exploratory study on Android API deprecation and identified that the Android framework is regularly cleaned-up from deprecated API and their maintainers ensure that deprecated APIs are commented to provide replacement messages. However, those APIs are not consistently annotated and documented and the existing documentation is not frequently updated. Wang et al. (Wang et al., 2020) conducted an exploratory study on Python library API deprecation and observed that API deprecation is poorly handled by library contributors and the usage of deprecated APIs is rarely changed. Yasmin et al. (Yasmin et al., 2020) proposed a framework to investigate RESTful API deprecation and revealed that 87.3% of breaking changes were not deprecated on previous versions. Many other researchers study how API evolves, measure breaking changes, and analyze their impact on client systems (Brito et al., 2020, Xavier et al., 2017). We notice that none of these cover the JavaScript ecosystem.

2.4 Comparison to Other Languages

The analysis of deprecation mechanisms and practices have been the main focus of previous studies on different programming languages, such as Java (Brito et al., 2018, Li et al., 2018, Sawant et al., 2016, Sawant et al., 2019), Python (Wang et al., 2020), C# (Brito et al., 2018) and Pharo/Smalltalk (Robbes et al., 2012). We first note that, different from JavaScript, Java, C# and Pharo/Smalltalk have built-in deprecation mechanisms. However, they implement different features to handle deprecation. Java provides two main ways to handle deprecated elements. Developers can annotate an element with *@Deprecation* to mark an obsolete API. It accepts two options to indicate the version the element was deprecated and specify whether or not the element will be removed in a future release. This mechanism automatically triggers

compilers to warn when the deprecated element is being used. Additionally, Javadoc[12] also provides an *@deprecated* annotation that helps generate API documentation. However, Java does not provide means to provide the deprecation severity or indicate what parts of a certain API is deprecated. Other studies have shown that code comments and documentation notes are also common deprecation strategies in the Java ecosystem (Sawant et al., 2018a). Also, Java API consumers do not appear to react to deprecation (Sawant et al., 2016).

Pharo/Smalltalk implements a deprecated method in which users can provide a message to be outputted at run-time as a warning. Also, after an API deprecation is introduced, a considerable number of projects do not update to newer versions (Robbes et al., 2012). C# has an *ObsoleteAttribute* that can be used to mark an attribute or method as deprecated. It can receive a message to be thrown by the compiler in compile-time. A second option can also be passed to treat the deprecation as an error. This option might be useful to indicate the severity of a deprecation (Brito et al., 2018). While there is no deprecation mechanism built-in in Python, developers can rely on the *deprecation decorator*[13] project hosted on PyPI, the official third-party software repository for Python. This decorator also logs deprecation warnings to consumers. However, library maintainers use different decorators or adopt ad-hoc local solutions. Alternative strategies, such as hard-coded warnings and comments, are common as well. Also, the usage of deprecated APIs is not usually addressed by developers during the evolution of Python projects (Wang et al., 2020).

The JavaScript deprecation strategies we found in our web research seem to be strongly influenced by deprecation mechanisms and features of other popular programming languages such as Java and C#. The JSDoc *@deprecated* annotation, for instance, is equivalent to the one defined by the Javadoc specification. Other mechanisms, such as warning messages and code comments, are also very similar to built-in features and code conventions found in other programming languages. Although Python is similar to JavaScript in terms of native deprecation mechanisms, since both have none, the Python community seems to be more inclined to build official deprecation tools.

---

[12] https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html
[13] https://pypi.org/project/Deprecated/

2.5 Final Remarks

In this chapter, we presented important information that is necessary to better understand this work. We first detailed the concepts of Application Programming Interfaces (APIs) and API deprecation, along with some examples and an explanation of why deprecation is an important practice for the software engineering research community and industry. Furthermore, we discussed the JavaScript role in the software development industry and how little discussion has been presented so far on JavaScript API deprecation. In addition, we presented previous works related to API deprecation and breaking changes. We also compared deprecation mechanisms commonly available in other popular programming languages. Unlike all previous studies discussed in this chapter, in this Master thesis we aim at providing a detailed understanding of depreciation practices in the JavaScript ecosystem. To the best of our knowledge, this is the first exploratory research to investigate the state of practice of API deprecation in JavaScript.

Chapter 3

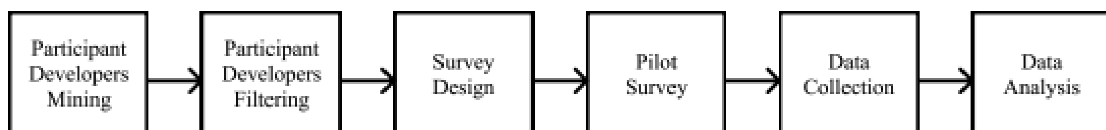
3 Survey with Developers on Deprecation Practices

Libraries provide reusable functionality to consumer applications through their APIs. However, APIs evolve over time and might be discontinued. In such cases, deprecation is recommended to indicate that the use of a certain API should be avoided. Unlike other popular programming languages, such as Java and C#, JavaScript provides no native deprecation mechanisms. In addition, we observed that there is no standard way for API deprecation in JavaScript.

In this chapter, we present a survey study aiming at understanding how developers deal with deprecated APIs in JavaScript packages. The goal is to provide a more thorough comprehension of which deprecation strategies are the most common among developers. In our study, we first investigate to what extent developers encounter deprecated APIs and deprecate them. We argue that a detailed study of commonly adopted deprecation strategies may contribute to the software engineering community by understanding the current state of practice of API deprecation in the JavaScript ecosystem.

In our study, we survey 109 developers who contributed to JavaScript open source projects in GitHub. We investigate what deprecation strategies they often see and preferably adopt. Section 3.1 presents our main goal and the research questions we designed. Section 3.2 details the strategy used to collect and identify eligible participants. Section 3.3 presents the survey design. Section 3.4 reports the results. We discuss the threats to validity in Section 3.5 and conclude this chapter in Section 3.6.

Figure 3.1 – Survey study methodology steps.



3.1 Goal and Research Questions

In order to better understand what deprecation strategies JavaScript developers often see and preferably adopt on their projects, and how they react to deprecation, we investigate three research questions:

- RQ1: To what extent do developers see deprecated APIs and deprecate in JavaScript packages?
- RQ2: What deprecation strategies in JavaScript do developers most commonly see and adopt?
- RQ3: What priority do developers give to deprecation issues?

3.2 Study Design

This section describes the methodology we followed to answer the research questions presented in the previous section. Figure 3.1 presents the steps we followed in the study. First, mine and filter developers do participate in the survey. Afterwards, we construct the survey and conduct a pilot to validate the proposed questions. Then, we run the survey to collect data for later analysis. Sections 3.2.1 and 3.2.2 describe those steps.

Mining Developers: We were interested in surveying active developers with recent contributions to JavaScript projects. We then mined GitHub users to obtain developer emails that match that specific profile. We started by using GitHub search API[14] to search for and randomly select developers with contributions to JavaScript projects on GitHub. We filtered out developers with less than 50 commits in the last year, as we could not ensure they have been actively working with JavaScript recently. We also removed developers with more than 100 followers, as very popular developers could be less likely to respond to surveys. We ended up with a list of 14,480 email addresses of active developers of GitHub projects.

Survey Design: Since we find little academic literature on JavaScript API deprecation, we relied on blogs, forums, and Q&A websites such as StackOverflow to get initial information on how developers deprecate JavaScript APIs. This initial research revealed common approaches and discussions that helped us create the survey questions. We also found third-party libraries specifically built to aid API deprecation in JavaScript, such as depd[15]. From these results, we created several questions targeting two perspectives: the API consumer (developers

[14] https://docs.github.com/en/rest/reference/searchsearch-users
[15] https://www.npmjs.com/package/depd

who use deprecated APIs) and the API provider (developers who maintain libraries and might deprecate APIs). We were interested in investigating what developers know about deprecation in JavaScript, which strategies they most commonly see or use, what priority do they give to deprecation issues and what are their thoughts in general related to deprecation practices in JavaScript.

As a result of several iterations, we formulated five closed-ended questions to capture both perspectives. We restricted the survey to five questions since we believed a small survey would encourage participants to respond more readily. Additionally, we added an open-ended question to encourage the respondents to add any thoughts, experiences, or suggestions regarding API deprecation in JavaScript.

We first conducted a pilot survey with nine developers to validate the proposed questions and evaluate if they are adequate and clear. That helped us minimize the risk of sending the survey to a large number of developers with unclear or ambiguous questions. After the pilot survey was conducted, we made minor copy adjustments to the questions' statements. We also added a "Documentation" option to Questions 2 and 5. This option was not initially considered and appeared in four answers. The pilot answers were not considered in the final results. Table 3.1 presents all survey questions. The five columns detail, respectively, the question number, the question statement, the type of the question, the options for the closed-ended questions, and the reasoning behind the question. Note that some questions are skipped depending on the answers of previous questions.

3.3 Survey Results

We sent 100 survey emails daily to developers until we reached at least 100 responses. We reached the goal when we sent the 1,400th email and had 109 responses, which corresponds to about 8% response rate. Table 3.2 summarizes the answers to the five closed-ended questions.

Table 3.1 – Summary of survey questions to JavaScript developers.

#	Question Statements	Type	Options	Reasoning
	API Consumer Perspective			
Q1	How often do you see deprecated APIs or a deprecation message in JavaScript?	Closed-ended	Never Occasionally Sometimes Often Always	Understand if they have ever seen deprecated APIs and how often they perceive it. If Never, it skips Questions 2 and 3.
Q2	What types of deprecation have you seen before?	Closed-ended	Console message Documentation JSDoc Code comment Prefixed element Other	Understand what known or unknown deprecation strategies they consume
Q3	How fast do you fix deprecation problems on projects you are working on?	Closed-ended	As soon as possible Only when I have time Only if it is necessary Do not usually fix	Learn how developers react to deprecation and whether they consider it a critical issue
	API Provider Perspective			
Q4	How often do you deprecate the APIs (such as functions, classes, properties, etc) you implemented in JavaScript?	Closed-ended	Never Occasionally Sometimes Often Always	Understand if they have ever deprecated APIs and how often they deprecate. If Never, it skips Question 5.
Q5	How do you deprecate the APIs you implemented in JavaScript?	Closed-ended	Documentation JSDoc Console message Code comment Utility Prefixed element Deprecation list Other	Understand what known or unknown deprecation strategies they use
Q6	Is there any thoughts, suggestions or experiences you would like to share about deprecation mechanisms and practices in JavaScript?	Open-ended		Collect additional experiences or suggestions they might have on API deprecation

The table is divided in two major columns, presenting the survey results related to API consumers and as API providers, respectively. Within each major column, the first two sub-columns present the question number, the question statement and the options. The last two sub-columns present the results in absolute numbers and percentages. We discuss the main results in Sections 3.3.1 and 3.3.2. Note that RQ1 results relate to survey questions 1 and 4 (Q1 and Q4), while RQ2 results relate to survey questions 2 and 5 (Q2 and Q5). Finally, RQ3 results relate to survey question 3 (Q3).

Table 3.2 – Survey closed-ended questions and answers.

	API Consumer Perspective	#	%		API Provider Perspective	#	%
Q1	**How often do you see deprecated APIs in JavaScript?**			Q4	**How often do you deprecate APIs you implement in JavaScript?**		
	Never	8	7.3%		Never	22	20.2%
	Occasionally	41	37.6%		Occasionally	58	53.2%
	Sometimes	34	31.2%		Sometimes	18	16.5%
	Often	21	19.3%		Often	8	7.3%
	Always	5	4.6%		Always	3	2.8%
Q2	**What types of deprecation have you seen before?**			Q5	**How do you deprecate APIs you implement in JavaScript?**		
	Console message	89	88.1%		Documentation	44	50.6%
	Documentation	75	74.3%		JSDoc	42	48.3%
	JSDoc	49	48.5%		Console message	40	46.0%
	Code comment	25	24.8%		Code comment	37	42.5%
	Prefixed element	23	22.8%		Utility	14	16.1%
	Other	6	6.0%		Prefixed element	14	16.1%
					Deprecation list	3	3.4%
					Semantic Versioning	3	3.4%
					Other	2	2.2%
Q3	**How fast do you fix deprecation problems?**						
	As soon as possible	25	24.8%				
	Only when I have time	34	33.7%				
	Only if it is necessary	37	36.6%				
	Do not usually fix	5	5.0%				

3.3.1 The API Consumer Perspective

This section discusses the main results of our survey study focusing on the first three questions, related to the API consumer perspective.

Q1. How often do developers see deprecated APIs in JavaScript projects?

Only 4.6% of the respondents always see deprecated APIs during development activities, while 19.3% notice them often. In contrast, 31.2% of the respondents see deprecated API only sometimes and 37.6% occasionally. Lastly, 7.3% have never seen deprecated APIs in JavaScript. From this result, it is interesting to observe that about 75% of developers do not see deprecated APIs very often, either sometimes, occasionally, or never. We argue that deprecated APIs in JavaScript are not common, or not informed by maintainers, or deprecation messages are not being perceived by developers. If the last one is true, it might imply that library maintainers need to communicate deprecation more effectively.

Q2. What deprecation mechanisms developers have seen before?

In our survey, the most common deprecation solutions mentioned by the developers were console messages (88.1%) and project documentation (74.3%). Also, 48.5% of the

developers have seen deprecated elements annotated with the JSDoc annotation @deprecated and 24.8% have noticed them in single code comments. Moreover, 22.8% have seen APIs prefixed with any sort of deprecation prefixes, such as deprecated__. Finally, six developers revealed other solutions: four developers mentioned they have seen deprecation console messages specifically during the package/library installation, one respondent indicated deprecation error messages at runtime, and one added the usage of the custom Deprecated<T> type in TypeScript. Overall, this result sug gests that, from the consumer perspective, deprecation messages delivered via console messages and project documentation are more likely to be perceived by developers. Deprecation communicated via code comments, including the JSDoc annotation as well, might be more appropriate for internal APIs.

Q3. How fast developers fix deprecation problems?

24.8% of the respondents state they fix deprecation issues as soon as possible, 33.7% fix only when they have time, and 36.6% fix only if necessary. Lastly, a minority of the developers (5%) do not usually fix deprecation issues. From this result, we observe that only 24.8% of developers treat deprecation issues urgently. This suggests that developers might not think that deprecation in JavaScript is important or worth migrating.

3.3.2 The API Provider Perspective

Q4. How often developers deprecate APIs in JavaScript projects?

Only 2.8% of the developers always deprecate APIs, while 7.3% often deprecate. While 16.5% deprecate sometimes, over a half (53.2%) deprecate APIs occasionally. Lastly, 20.2% have never deprecated APIs. Overall, 90% of developers do not deprecate APIs very often. We observe that most developers do not maintain external APIs in the JavaScript ecosystem or do not have the need to deprecate them.

Q5. What deprecation mechanisms developers use to deprecate API?

The majority of the API providers (50.6%) use the project documentation to inform about deprecated APIs. Next, we find three categories with similar ratios: 48.3% annotate deprecate elements with the JSDoc annotation @deprecated; 46% use console messages to warn about deprecated API; and 42.5% add code comments next to API to indicate deprecation. Moreover, 16.1% of the developers use utilities to aid API deprecation and 16.1% prefix API elements to indicate they have been deprecated. Lastly, 3.4% maintain, somewhere within the project, a list/object of deprecated elements, while 3.4% state they remove deprecated APIs on

major releases, following the Semantic Versioning specification[16]. One developer added they rely on the npm-deprecate[17] npm CLI command to deprecate the whole package and another one indicated the usage of the custom Deprecated<T> TypeScript type.

Deprecation communicated via documentation and console messages seem to be similarly popular in both consumer and provider perspectives. However, we observe JSDoc annotation and code comment are similarly common strategies among provider developers, but not seen as often by consumer developers. This reinforces our argument that JSDoc annotation and code comment might not be very effective to inform consumers about deprecated APIs.

3.3.3 Developers' Further Insights

In the last question, we encourage participants to share thoughts on JavaScript deprecation. In this part, we received 20 answers.

Overall, three developers argued that excessive deprecation logging can annoy developers while working. One developer claimed that when a deprecation occurrence does not look critical, they tend to perceive it as unnecessary. The current state of practice of deprecation in which maintainers tend to retain deprecated APIs in favor of clients was criticized by three developers. In this case, one developer noted: "*the current practice/implementation of deprecation in JS breeds a culture of complacency on top of old and dangerous systems*", causing an endless backward compatibility effort that degrades code health. As a solution to this problem, three developers advised to deprecate more often and retain less deprecated APIs. It was noted that breaking changes should not be avoided if the project is following Semantic Versioning best practices.

One developer stated that some deprecation messages come from transitive dependencies. In such cases, there is no direct action to address the deprecation. Moreover, one developer indicated that, with the fast and ever-evolving JavaScript ecosystem, "*deprecation of API is time-consuming and laborious*", but it is nevertheless beneficial. Also, the more visible and persistent a deprecation warning is, the more likely developers are to address it. Developers also emphasized that deprecation communication does not necessarily need to happen at the API level, since major releases are expected to bring breaking changes. For example, one developer stated: "*when we roll out a new major version of a library, the changes are always*

[16] https://semver.org
[17] https://docs.npmjs.com/cli/deprecate

breaking". In those cases, any required upgrade should preferably be communicated on release notes.

As package maintainers, two developers suggested that console messages are the most efficient way to communicate deprecation since they believe developers always have their eyes on the console while coding. However, as package consumers, developers presented that they would like to be able to suppress deprecation messages as they wish, even if temporarily. Additionally, three developers emphasized the importance of clear and constant communication about deprecated APIs. For example, one developer stated: "*for any consumed APIs, as much deprecation communication as possible is preferred*", either with an internal team member or with client systems. Finally, four developers suggested a cohesive deprecation strategy as an appropriate way to approach deprecation on a project: "*your versioning strategy is the way you inform your consumers what is the scope of a change via the version number"*.

To conclude, we present five deprecation best practices suggested by a developer:

1. Plan a deprecation strategy and make clients aware of it;
2. Release a minor version with the deprecated API;
3. Inform clients about upcoming changes via project documentation and console logs, preferably with a message containing a target date and release and a link to a migration guide;
4. Release a major version with breaking changes, along with a release note containing a link to the migration guide;
5. Though it is not recommended, if a deprecated API needs to be retained, either add a UNSAFE__ or similar prefix or provide them through an opt-in flag, such as –legacy or –insecure.

3.4 Threats to Validity

The qualitative study presented in this chapter has some limitations that could potentially threaten our results, as we explain next. The first threat to validity of this study is related to target developers. This survey study findings cannot be directly generalized since the participants might not be representative of the general population of JavaScript developers outside of GitHub. However, GitHub is the most popular software development platform, for both public and private projects. Future work replications on this topic should address these issues.

Second, the survey questions might have been unclear or ambiguous for participants. To minimize this thread we conducted several iterations of reviews when formulating the questions. We also conducted a pilot survey to collect feedback and improve the survey. Third, developers might have provided unreliable or unrealistic answers. For example, the provided deprecation strategies or reactions might deviate from reality. To minimize this threat, we did our best to send a short and focused survey to developers. We also informed participants that the research had academic purposes only.

Furthermore, the deprecation mechanism options used in the survey might bias respondents by limiting their ideas of what deprecation looks like. However, three researchers verified the categories and we also provided an "other" option to encourage respondents to add other solutions. Also, the experiment observations and analysis were conducted by the author manually and therefore they may contain misunderstandings. However, the results were evaluated by the first author and validated by the co-authors.

Finally, the experimental observations and analyses were manually conducted by the authors of the paper, therefore, the results might be subjective to authors' bias. To mitigate this threat, we adopted thematic analysis to analyze survey results.

3.5 Final Remarks

In this chapter, we described a survey study with developers to understand what API deprecation strategies are most commonly present in JavaScript projects and libraries. We found that there is no standard preferable strategy to deprecate JavaScript APIs. Overall, the most commonly adopted deprecation mechanisms are console message, project documentation, JSDoc annotation, and code comment. Developers usually learn about deprecated JavaScript APIs via console message and project documentation. Additionally, most JavaScript developers (70%) only address deprecation issues if necessary or if time permits. Furthermore, we presented an extensive analysis of developers general thoughts on the current state of practice of JavaScript deprecation, along with recommended approaches for deprecating APIs. Developers suggested that planning a deprecation strategy and making client aware of it is an efficient way to handle deprecation in a JavaScript project. Additionally, as much clear and consistent depreciation communication as possible is preferred, either via deprecation messages or project documentation. Furthermore, respondents advised to retain less deprecated APIs in favor of code health. However, if they need to be maintained, either add UNSAFE__ or similar prefix to a deprecated API or provide them through an opt-in flag, such as –legacy or –insecure.

In the next chapter of this dissertation, we investigate API deprecation practices in the JavaScript ecosystem by means of a mining study. We turn our attention to popular JavaScript libraries and analyze which deprecation mechanisms are most commonly present on their source code. We also analyze to what extent those mechanisms are consistent among and within JavaScript packages.

Chapter 4

4 Mining API Deprecation in JavaScript Packages

Software packages expose APIs to provide reusable functionality. However, APIs evolve over time and might be discontinued or promote breaking changes to their consumer applications. In such cases, it is recommended that package maintainers communicate via deprecation messages that the use of a certain API should be avoided or updated. Unlike other popular programming languages, such as Java and C#, JavaScript provides no native deprecation mechanisms. In the previous chapter, we conducted a survey study that suggested that there is no standard approach to deprecate APIs in JavaScript. However, participant developers indicated four mechanisms that are most commonly used: *deprecation utility, code comment, JSDoc annotation and console message*.

In this chapter, we present a quantitative mining study aiming at analyzing API deprecation occurrences in popular JavaScript libraries. The goal is to provide a deeper understanding of which deprecation mechanisms are most commonly present in popular JavaScript packages and to what extent deprecation strategies are consistent among and within JavaScript libraries. In our study, we mine the source code of the 320 most dependent upon JavaScript packages on npm and search for deprecation occurrences that match one of the top four mechanisms developers indicated on the survey. Section 4.1 presents our main goal and the research questions we designed for this study. Section 4.2 details the study design, including our source code mining strategy. Section 4.3 reports the results. We discuss the threats to validity in Section 4.4 and conclude this chapter in Section 4.5.

4.1 Goal and Research Questions

In order to better understand which deprecation mechanisms are most commonly present in popular JavaScript packages, and to what extent deprecation mechanisms are consistent among and within JavaScript packages, we investigate two research questions:

- RQ1: What deprecation API mechanisms are the most common in popular JavaScript packages? We investigate four common deprecation mechanisms, according to developers, and analyze how they are actually implemented in the source code of popular JavaScript packages.

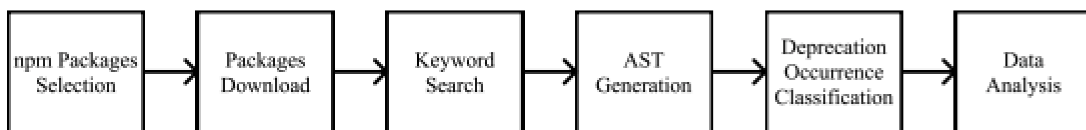
- RQ2: To what extent are deprecation strategies consistent in popular JavaScript packages? We analyze if the deprecation mechanisms mined are consistently adopted in a package or among all packages, and if they are used alone or combined.

4.2 Study Design

This section describes the methodology we followed to answer the research questions presented in Section 4.1. Figure 4.1 presents the steps we followed in the study. First, we present how we selected the JavaScript packages to compose the data set we used. Then, we downloaded the selected packages. Afterwards, we describe the search strategies we adopted to find API deprecation occurrences in the target libraries. Next, we detail how we identified deprecation occurrences through abstract syntax trees. Then, we classified all deprecation occurrences found into one of four deprecation mechanisms. Finally, we conducted the collected data analysis.

Selecting Candidate JavaScript Packages: In order to answer our research questions, we were interested in analyzing popular JavaScript libraries and investigating which deprecation mechanisms they used to deprecate their APIs.

Figure 4.1 – Mining study methodology steps.



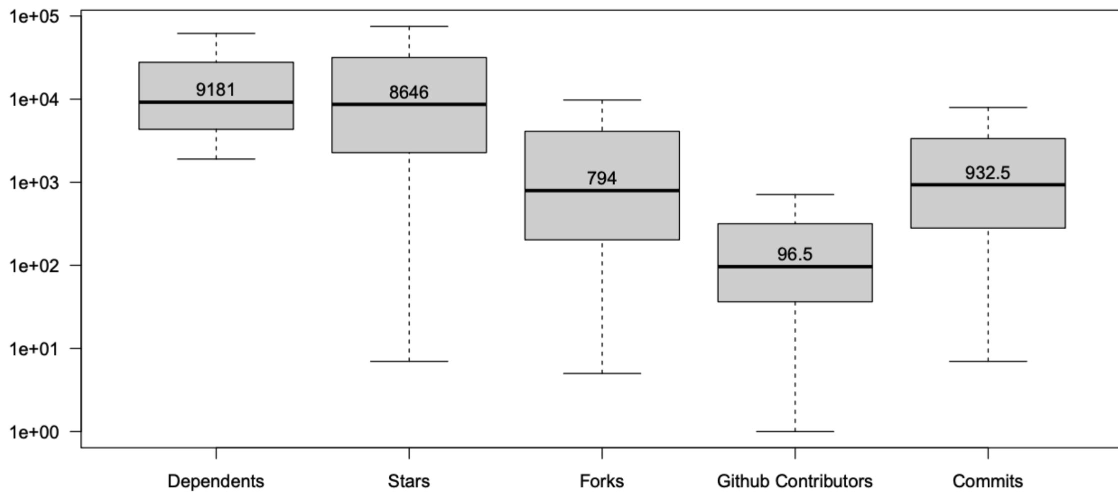
Thus, we first selected the top-320 most depended upon packages according to the npm registry[18] to compose our library data set. npm is the largest and most popular package manager and repository for JavaScript applications. Therefore, the npm registry website is an indicator of package popularity and their amount of client applications. To identify characteristics of these JavaScript packages, we also collected metrics from their GitHub repositories: stars, forks, contributors, and commits.

Figure 4.2 shows those statistics about these packages based on npm and GitHub data retrieved in July, 2021. In particular, it presents box plots with the number of dependent clients, stars, forks, GitHub contributors and commits. We collected the first metric from npm, while

[18] https://www.npmjs.com/browse/depended

the others were obtained from GitHub. As can be observed in this figure, the selected packages are not only highly popular (e.g., median of 8.6K stars), but also forked a lot. They are also active and have thousands of dependent clients. For example, the top-3 libraries have 280K, 235K, and 172K dependents, respectively.

Figure 4.2 – JavaScript packages characteristics: npm dependents and GitHub statistics.



Searching for Deprecation Occurrences: We downloaded the source code of the 320 selected packages, considering their latest stable version in March 2021. We then searched for all occurrences of the substring *deprecat* on JavaScript files to find possible deprecation candidates. We also tried to find deprecation occurrences by using the keyword *obsolete*, but we only found it in 18 files and none of them were deprecation occurrences. Thus, we focus our research on the most used term for depreciation (*deprecat*). While navigating through package files, we only considered main source code files, excluding test, minified, and non-JS files (e.g., CSS and HTML). Next, we use Flow[19], a well-known JavaScript code parsing library maintained by Facebook, to parse each file containing deprecation candidates, and to generate their corresponding abstract syntax trees. An abstract syntax tree (AST) is a tree representation of the source code structure written in a programming language. Each node of the tree has a type and represents a language syntax occurring in the source code. By analyzing ASTs, we are able to programmatically detect, for instance, code comments and their content, function declarations and their identifier names, expression calls and their components, and many other components of code structure and language syntax. Listing 4.1 shows an example of an AST.

[19] https://flow.org

If we look at lines 5 to 15, we can note that a function named power is being declared with one parameter named base. In addition, lines 41 to 45 describes a code comment, of type line, starting and ending on line 1, containing Returns the base to the power of two. From this tree structure, we are able to programmatically detect JavaScript constructs and determine deprecation occurrences.

Based on the most common mechanisms indicated by developers in our survey study (Chapter 3), we focused on finding and automatically categorizing deprecation occurrences of 4 types. Listing 4.2 presents code snippets of each one of those deprecation solutions: deprecation utility, code comment, JSDoc annotation, and console message. We used the ASTs obtained from the analyzed JavaScript files to automatically find occurrences of deprecation, based on the matching rules for each category described as follows:

1. Deprecation utility: any function declaration or call in which the function identifier name matches the substring *deprecat*, as demonstrated in Listing 4.2;
2. Code comment: any type of code comment includes matches the substring *deprecat*, as demonstrated in Listing 4.2, excluding occurrences of JSDoc annotations;
3. JSDoc annotation: the exact usage of the JSDoc *@deprecated* annotation inside a comment, as demonstrated in Listing 4.2;
4. Console messages: calls of any console function - such as warn, log, error - in which the message argument is a string literal that matches the substring *deprecat*, as demonstrated in Listing 4.2.

We manually evaluated samples of each category to measure the precision of our script to correctly identify API deprecation (each sample size ensured a confidence level of 95% and a confidence interval of 5%). Each deprecation case was evaluated by the author of the dissertation and validated by the supervisors. In case of conflict, all researchers discussed until an agreement was reached. In this preliminary evaluation, we find a precision of 98% for deprecation utility, 81% for code comment, 100% for JSDoc comment, and 100% for console message. This way, our tool can be used with a good level of confidence.

```
{
    "type": "Program",
    "loc": { ... },
    "body": [{
        "type": "FunctionDeclaration",
        "loc": { ... },
        "id": {
            "type": "Identifier",
            "loc": { ... },
            "name": "power"
        },
        "params": [{
            "type": "Identifier",
            "loc": { ... },
            "name": "base"
        }],
        "body": {
            "type": "BlockStatement",
            "loc": { ... },
            "body": [{
                "type": "ReturnStatement",
                "loc": { ... },
                "argument": {
                    "type": "BinaryExpression",
                    "loc": { ... },
                    "operator": "*",
                    "left": {
                        "type": "Identifier",
                        "loc": { ... },
                        "name": "base"
                    },
                    "right": {
                        "type": "Identifier",
                        "loc": { ... },
                        "name": "base"
                    }
                }
            }]
        }
    }],
    "comments": [{
        "type": "Line",
        "loc": {
            "start": {
                "line": 1, "column": 0
            },
            "end": {
                "line": 1, "column": 39
            }
        },
        "value": " Returns the base to the power of two"
    }]
}
```

Listing 4.1 – Example of AST generated from code snippet.

```
1  ——— Deprecation utility example ———
2  const deprecate = require("depd")("my—math—module");
3
4  function addDigits(x, y) {
5      deprecate('addDigits is deprecated. It will be deleted on version 3.0.')
6      return x + y;
7  }
8
9  ——— Code comment example ———
10 function addDigits(x, y) {
11     // Function deprecated since version 2.3. It will be deleted in version 3.0.
12     return x + y;
13 }
14
15 ——— JSDoc annotation example ———
16 /**
17  * @deprecated since version 2.3. It will be deleted on version 3.0.
18  */
19 function addDigits(x, y) {
20     return x + y;
21 }
22
23 ——— Console message example ———
24 function addDigits(x, y) {
25     console.warn("addDigits is deprecated. It will be deleted on version 3.0.");
26     return x + y;
27 }
```

Listing 4.2 – Examples of JavaScript deprecation approaches.

4.3 Study Results

We observed deprecation occurrences on 122 (38%) out of the 320 analyzed packages. Considering those 122 packages, we found 2,501 deprecation occurrences in 681 (˜2%) out of 35,318 files.

Figure 4.3a presents the deprecation occurrences by category. The most frequent deprecation mechanism is *deprecation utility* (41.7% of the cases), which represents any sort of code function specially written to support deprecation. This category is followed by *code comment* (34.5%) and *JSDoc annotation* (18.8%). Lastly, the direct usage of *console messages* is the least common (4.7%). It is important to note that, although *deprecation utility* accounts for almost half of deprecation occurrences present in subject packages, fewer packages adopt this strategy when compared to *code comments*. For instance, package *@alifd/next* comprises 41.4% (432) of the 1,044 deprecation utility occurrences. We believe that this explains why fewer developers indicated the usage of deprecation utilities as opposed to other strategies in our survey results described in Chapter 3.

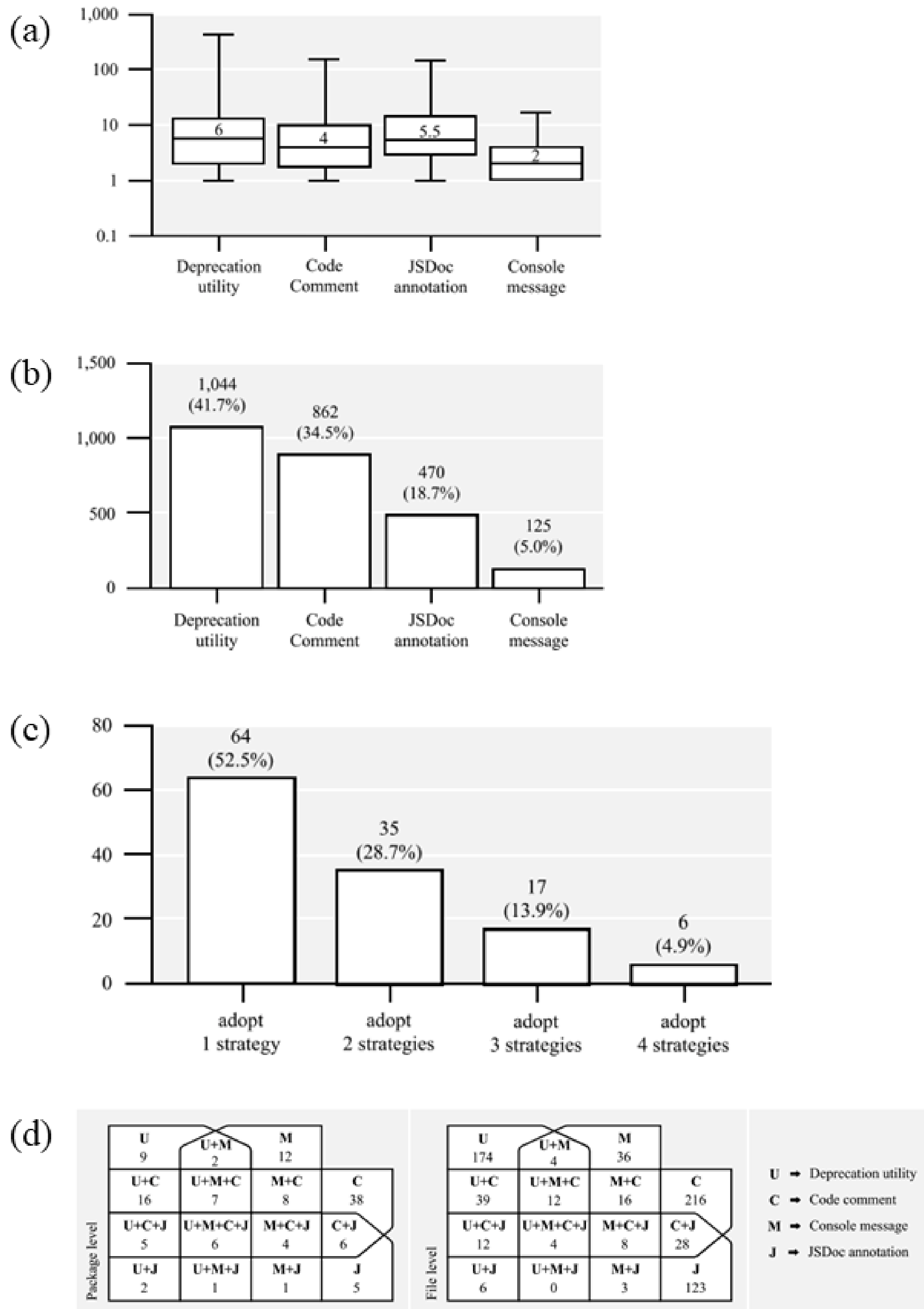
Figure 4.3b presents the distribution of the deprecation per package. The median values range from 2 (*console message*) to 6 (*deprecation utility*). Notice that a small number of

packages concentrate a large number of occurrences, particularly in the case of deprecation utilities. Indeed, packages that adopt *deprecation utility* tend to have more deprecation occurrences and may have specific deprecation needs that are not satisfied by other simpler mechanisms.

> *RQ1: What deprecation API mechanisms are the most common in popular JavaScript packages?* There is no standard approach to deprecate JavaScript APIs. However, we find that deprecation utility is the most frequent solution (41.7%), followed by code comment (34.5%), JSDoc annotation (18.8%) and console message (4.7%).

Next, Figure 4.3c summarizes the combination of deprecation strategies per package. Around 52% of the analyzed packages adopt only one deprecation mechanism, while 28.7% combine two deprecation mechanisms. As we increase the number of combined mechanisms, the number of packages decreases. For example, we only found the occurrence for the four studied mechanisms in 6 (4.9%) packages. Figure 4.3d presents a detailed view of these data: a Venn diagram showing the intersection of deprecation mechanisms. This detailed view is presented in two levels of granularity: package and file level. In both package and file levels, the most adopted single strategy is *code comment* (38 packages and 216 files). The most common combination is *deprecation utility* and *code comment*, which is present in 16 packages and 39 files. Although they are not adopted by many packages, *deprecation utility* and *JSDoc annotation* are highly used at the file level, with 174 and 123 occurrences, respectively. This suggests that packages that adopt those two strategies tend to use those mechanisms very frequently. Also, similarly to what occurs at the package level, most files implement a single deprecation mechanism. The two most frequent combinations at file level are *deprecation utility* and *code comment* (39 files) and *code comment* and *JSDoc annotation* (28 files).

Figure 4.3 – API deprecation mechanism occurrences in JavaScript packages. (a) Deprecation occurrences by category. (b) Occurrences distribution by category. (c) Number of packages by the number of deprecation strategies adopted. (d) Number of packages and files that adopt mechanism combinations.



In summary, we find no standard solution to deprecate JavaScript APIs. Moreover, we observe that the four studied deprecation strategies (deprecation utility, code comment, JSDoc annotation, and console message) are used both standalone or combined at package and file levels.

RQ2: To what extent are deprecation strategies consistent in popular JavaScript packages? JavaScript deprecation mechanisms might be used alone or combined at packages and file levels. Over half of the analyzed packages (52.5%) adopt only one deprecation mechanism, while the remaining part combines two or more deprecation strategies. The most adopted single strategy is *code comment*, whereas the most common combination is *deprecation utility* and *code comment*.

4.4 Threats to Validity

The quantitative study presented in this chapter has some limitations that could potentially threaten our results, as we explain next. First, we focused the analysis on 320 JavaScript open-source packages hosted in npm, the most popular JavaScript package manager. Despite these observations, our findings cannot be generalized to other systems implemented in other languages or closed-source packages. Additionally, we analyzed packages with a large number of dependent clients, as we expect them to be examples of well-maintained packages and representative case studies of open-source packages with many dependent clients. However, their maintainers may not represent the whole population of JavaScript developers. However, GitHub is the most popular software development platform, for both public and private packages. Future replication work on this topic could be conducted to address these issues.

Second, to identify deprecation occurrences, we only searched for matches of *deprecat*. We tried to find other occurrences by using the keyword *obsolete*, but only 18 out of 35,318 files were found and none of them indicated deprecation. Thus, we focused on *deprecat* occurrences. Although being deliberate, this choice might have caused us to miss cases in which other terms are used. Furthermore, since we mine all JavaScript files, the deprecation strategies we analyzed might also be related to internal APIs that are not visible to consumers. Future studies that select only external APIs should address this issue. Moreover, the JavaScript tool for the mining study was implemented upon Flow, a well-known JavaScript code parsing library maintained by Facebook and, thus, the risk of errors is reduced. Additionally, we have manually

inspected its output (each sample with a confidence level of 95% and a confidence interval of 5%). We find a precision of 98% for deprecation utility, 81% for code comment, 100% for JSDoc comment, and 100% for console message. Thus, our script can be used with a good level of confidence. Finally, the categorization of the deprecation occurrences we mined is subjected to the author/interpreter bias, although other members of our group verified the categories.

4.5 Final Remarks

In this chapter, we presented an empirical mining study regarding API deprecation in the JavaScript ecosystem. This work can help developers better understand JavaScript API deprecation approaches and offer guidance on which mechanisms are more appropriate to a certain package context.

We downloaded the top 320 popular JavaScript packages on npm and analyzed their source code to identify API deprecation occurrences. After investigating API deprecation occurrences on those packages, our results suggest that there is no standard approach to deprecate JavaScript APIs and there is no consistency in implementing a deprecation strategy. However, we find that deprecation utility is the most frequent solution (41.7%), followed by code comment (34.5%), JSDoc annotation (18.8%) and console message (4.7%). Additionally, we find that those deprecation mechanisms might be used alone or combined at packages and file levels. Over half of the analyzed packages (52.5%) adopt only one deprecation mechanism, while the remaining part combines two or more deprecation strategies. The most adopted single strategy is *code comment*, whereas the most common combination is *deprecation utility* and *code comment.*

In the next chapter of this dissertation, we take a step further on this investigation and analyze how API deprecation evolves overtime. We continue analyzing popular JavaScript libraries and discuss how API deprecation mechanisms change between releases. As a result, we identify increasing and decreasing trends and investigate when deprecation is usually introduced.

Chapter 5

5 Analysis of Deprecation Evolution

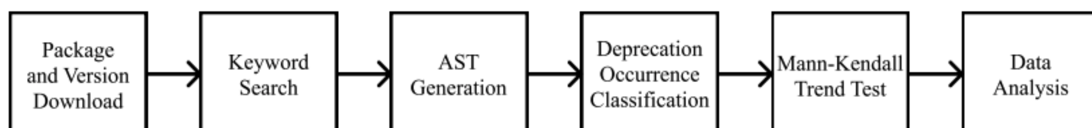
Software packages expose APIs to provide reusable functionality. However, as software systems evolve, APIs evolve as well, and thus might be discontinued or promote breaking changes to client applications. In such cases, it is recommended that package maintainers communicate via deprecation messages that the use of a certain API should be avoided or updated. Unlike other popular programming languages, such as Java and C#, JavaScript provides no native deprecation mechanisms.

In the previous chapter, we conducted a mining study to analyze API deprecation occurrences in popular JavaScript packages. Results suggest that there is no standard approach to deprecate JavaScript APIs and there is no consistency in implementing a deprecation strategy. Additionally, we find that the analyzed deprecation mechanisms (deprecation utility, code comment, JSDoc annotation, and console message) might be used alone or combined at package and file levels. In this chapter, we investigate how deprecated APIs evolve over the lifetime of third-party packages. We mine the source code of the 50 most-dependent upon JavaScript packages on Libraries.io and analyze how the number of deprecated APIs increase and decrease between version releases. Section 5.1 presents our main goal and the research questions we propose for this study. Section 5.2 details the study methodology, including our source code mining strategy and history analysis algorithm. Then, section 5.3 reports the results. Finally, we discuss the threats to validity in Section 5.4 and conclude this chapter in Section 5.5.

5.1 Goal and Research Questions

In order to better understand how deprecated APIs in JavaScript evolve over time in third-party packages, and when those APIs are added or removed, we investigate two research questions:

Figure 5.1 – Survey study methodology steps.



- RQ1: Do deprecated APIs increase or decrease overtime in JavaScript packages? We investigate common deprecation mechanisms in JavaScript and analyze how deprecated API changes over the lifetime of popular packages, i.e, if they present upward or downward trends.
- RQ2: Are deprecated APIs usually introduced and removed in major or minor releases? Study results from Chapter 3 suggest that deprecation should be introduced on minor releases and removed on major breaking releases. We investigate to what extent this recommendation is followed in popular JavaScript packages.

5.2 Study Design

This section describes the study design we followed to answer the research questions presented in the previous. Figure 5.1 presents the steps we followed in the study. We first present how we selected the JavaScript packages and their eligible versions to compose the data set for the study. Next, we describe the deprecation search and classification strategies through abstract syntax trees. Afterwards, we detail how we use the Mann-Kendall Trend Test to determine if there is an upward, downward or no deprecation trend between packages versions. Finally, we analyze the data collected to answer the study research questions.

Selecting Candidate JavaScript Package Versions: In order to answer the research questions proposed in this study, we were interested in analyzing popular JavaScript packages and investigating how deprecation evolved over time and when deprecated APIs are usually introduced or removed. We start by selecting the top-50 JavaScript packages sorted by the number of dependents on Libraries.io. Libraries.io[20] is a popular discovery service that indexes data from several package managers. They track package releases, project's code, dependencies and other useful information about open-source projects from a wide variety of programming languages. Thus, Libraries.io is a good data source for highly dependent JavaScript packages. We used the RESTful API provided by Libraries.io to list the top-50 JavaScript packages with the most dependents count thought the endpoint https://libraries.io/api/search?api_key=API_KEY&languages=JavaScript&order=desc&sort=dependents_count&per_page=50&page=1.

[20] https://libraries.io/platforms

Figure 5.2 – Analyzed JavaScript packages characteristics.

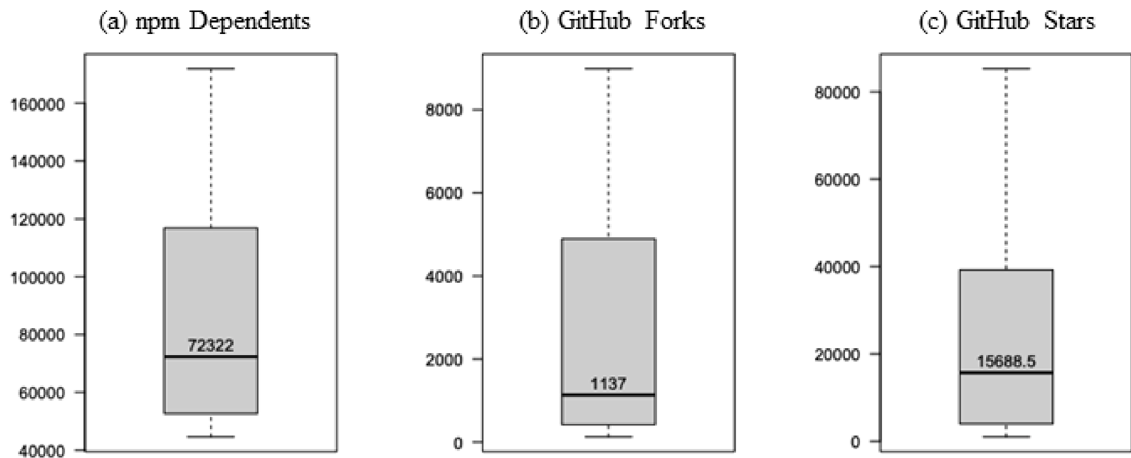


Figure 5.2 shows some statistics about the selected packages, retrieved in July, 2021, from Libraries.io. It presents box plots with the number of npm dependents in figure 5.2a, GitHub stars in figure 5.2b, and GitHub forks in figure 5.2c. As can be observed in this figure, the selected projects are very popular (e.g., median of 15.7K stars) and forked many times. In terms of dependent clients, the top-3 libraries have 280K, 235K, and 172K dependents, respectively.

For each package, we downloaded all versions, considering their latest patch releases. We also removed unstable versions, such as those made available from alpha and beta releases. We took this approach to remove possible inconsistencies on the number of deprecated APIs between releases. Table 5.1 details all selected packages for this study, their names and number of versions. Note that they are all very popular packages and most of them have a considerable amount of releases.

Searching for Deprecation Occurrences: We downloaded the source code of all versions from the 50 selected packages, up to their latest stable version in July 2021.

Table 5.1 – List of selected packages and number of versions.

#	Package Name	Number of Versions	#	Package Name	Number of Versions
1	eslint	146	26	webpack-dev-server	45
2	mocha	66	27	husky	36
3	webpack	132	28	style-loader	26
4	react	38	29	cross-env	11
5	chai	31	30	sinon	53
6	babel-core	41	31	vue	18
7	babel-loader	19	32	request	87
8	babel-eslint	18	33	eslint-config-prettier	55
9	lodash	48	34	commander	44
10	eslint-plugin-import	52	35	@babel/cli	13
11	@babel/core	15	36	sass-loader	29
12	react-dom	27	37	file-loader	21
13	prettier	36	38	fs-extra	44
14	rimraf	10	39	vue-template-compiler	8
15	babel-cli	19	40	nyc	58
16	@babel/preset-env	15	41	babel-preset-react	10
17	babel-preset-es2015	13	42	axios	21
18	css-loader	43	43	eslint-plugin-prettier	14
19	chalk	15	44	html-webpack-plugin	55
20	babel-preset-env	9	45	prop-types	5
21	eslint-plugin-react	75	46	coveralls	16
22	webpack-cli	18	47	eslint-plugin-jsx-a11y	25
23	gulp	24	48	core-js	36
24	rollup	154	49	moment	38
25	express	48	50	lint-staged	38

Next, we searched for all occurrences of the substring *deprecated* JavaScript files to find possible deprecation candidates in all package versions. We only considered main source code files, excluding test, minified, and non-JS files (e.g., CSS and HTML). Next, we use Flow[21], a well-known JavaScript code parsing library maintained by Facebook, to parse each file containing deprecation candidates, and to generate their corresponding abstract syntax trees (ASTs). Based on the most common strategies indicated by developers on our survey study (Chapter 3), we focused on finding and automatically categorizing, from the generated ASTs, deprecation occurrences of 4 types: deprecation utility, code comment, JSDoc annotation, and console message. For each package, we count how many deprecated APIs each version has. Additionally, we check if the amount of deprecated APIs increase or decrease between releases and, if either case happens, in which type of release it occurred (major or minor). Identifying Deprecation Trends: Afterwards, using the historical amount of deprecated APIs for each package, we use the Mann-Kendall Trend Test (MK Test) to verify if there is any upward or downward trend on the deprecation occurrences for each package. The MK Test evaluates whether a set of historical values tend to increase or decrease over time. It is based on a non-parametric form of monotonic trend regression analysis. To perform a MK test, we compute

[21] https://flow.org

the difference between a later value and all earlier values, $(y_j - y_i)$, where j>i, and assign 1, 0, or –1 to positive differences, no differences, and negative differences, respectively. Then, calculate the statistical test, S, from the sum of all those integers, as follows:

$$S = \sum_{i=1}^{n-1} \sum_{n}^{j=i+1} sign(y_j - y_i)$$

A large positive S suggests an upward trend, while a large negative S suggests a downward trend. When S is small, no trend is indicated. The statistical test τ, which has a range of –1 to +1, can be computed as:

$$\tau = \frac{S}{n(n-1)/2}$$

The null hypothesis of no trend is rejected when S and τ are significantly different from zero. To perform the trend test, we used the ***pymannkendall*** Python package, which implements the MK Test (Hussain and Mahmud, 2019).

5.3 Results

We observed deprecation occurrences on 32 (64%) out of the 50 analyzed packages. Additionally, we analyzed 1918 versions. Figure 5.3 presents two box plots represent ing the distribution of the amount of versions per package. Boxplot 5.3a shows the distributions of the amount of versions from all analyzed packages. Moreover, boxplot 5.3b shows the distributions of the amount of versions from packages with deprecation occurrences. We observe that packages with deprecation occurrences tend to have more releases.

Table 5.2 presents the deprecation trend results of the analyzed packages. The first column indicates the trend results for each package. The second and third columns describe the package names and the number of analyzed releases. In the last column, we display line plots representing the evolution in the number of deprecated APIs. We found 18 packages (36%), out of 50, with no deprecation occurrences (shown as *No Deprec.*). From 32 packages with deprecation occurrences, 22 (69%) packages present statistically significant trends (p-value > 0.05): 19 (59.4%) suggest an upward trend, while 3 (9.4%) packages indicate a downward trend. Finally, 10 (31.2%) packages present no statistically significant trend.

In addition to the overall number of deprecated over time, we also investigated how specific deprecation mechanisms evolve. In particular, we consider the deprecation mechanisms code comment, JSDoc annotation, console message, and deprecation occurrence. As summarized in Table 5.3, the first column presents the deprecation mechanisms, while the second, third, and fourth columns present the trend results. The total of packages in the last row is greater than the sum of packages with deprecated APIs (32) because the same package might adopt more than one mechanism.

Figure 5.3 – Versions in packages.

(a) Distribution of versions per package

(b) Distribution of versions per package with deprecation occurrences

Considering the deprecation *code comment*, we found 29 packages with deprecation occurrences. From those, 17 packages (58.7%) present an increasing trend in the usage of this mechanism, while 5 (17.2%) show a decreasing trend. Regarding the deprecation *JSDoc annotation*, we detected 8 packages with deprecation occurrences: 5 (62.5%) with an upward trend and 1 (12.5%) with a downward trend. For *console messages*, there were 17 packages with deprecation occurrences: 4 (23.5%) with an upward and 5 (29.4%) with a downward trend. Finally, we detected 20 packages with *deprecation utility*: 14 (70%) with an upward trend.

Table 5.2 – Mann-Kendall Trend Test results of deprecation occurrences on analyzed packages.

Trend	Package Name	# Releases	Evolution Plot
Upward	babel-loader	19	
	chai	31	
	commander	44	
	eslint	146	
	eslint-config-prettier	55	
	eslint-plugin-import	52	
	eslint-plugin-react	75	
	express	48	
	gulp	24	
	lint-staged	38	
	mocha	66	
	moment	38	
	prettier	36	
	react-dom	27	
	request	87	
	rollup	154	
	sinon	53	
	vue	18	
	webpack	132	
Downward	@babel/preset-env	15	
	lodash	48	
	webpack-cli	18	
No Trend	axios	21	
	babel-core	41	
	babel-preset-env	9	
	core-js	36	
	fs-extra	44	
	html-webpack-plugin	55	
	prop-types	5	
	react	38	
	vue-template-compiler	8	
	webpack-dev-server	45	
No Deprec.	@babel/cli	13	
	@babel/core	15	
	babel-cli	19	
	babel-eslint	18	
	babel-preset-es2015	13	
	babel-preset-react	10	
	chalk	15	
	coveralls	16	
	cross-env	11	
	css-loader	43	
	eslint-plugin-jsx-a11y	25	
	eslint-plugin-prettier	14	
	file-loader	21	
	husky	36	
	nyc	58	
	rimraf	10	
	sass-loader	29	
	style-loader	26	

Table 5.3 – Mann-Kendall Trend Test results of deprecation occurrences on analyzed, considering individual deprecation mechanisms.

Deprecation Mechanism	Upward Trends	Downward Trend	No Trend
Code Comment	17 (58.7%)	5 (17.2%)	7 (24.1%)
JSDoc Annotation	5 (62.5%)	1 (12.5%)	2 (25.0%)
Console Messages	4 (23.5%)	5 (29.4%)	8 (47.1%)
Deprecation Utility	14 (70.0%)	0 (0.0%)	6 (30.0%)
Total	40	11	23

RQ1: Do deprecated APIs increase or decrease overtime in JavaScript packages: Close to 60% of the analyzed packages present an increase in the number of deprecated APIs, while only 9.4% show decreasing trends. In particular, 70% of the packages with *deprecation utility* present upward trends. On the other hand, the deprecation mechanism with higher downward trends is *console message* (29.4% of the packages).

Study results from Chapter 3 emphasized recommendations for introducing depreciation on minor releases and removing them on major breaking releases, following Semantic Versioning. Table 5.4 presents the number of major and minor releases launched among the analyzed packages, and the number of increased and decreased deprecated APIs. In total, we identified 127 major releases. In those releases, packages increased their number of deprecated APIs 57 times and decreased in 19 cases. Thus, we have an increase ratio of 0.45 and a decrease ratio of 0.15 deprecated API by major release. When we look at minor releases, we observe 1,357 launches. In those minor releases, they increased their number of deprecated APIs 1,246 times and decreased 718 times. Hence, we have an increase ratio of 0.92, and a decrease ratio of 0.53 deprecated API by a minor release. These results reveal that different from what the JavaScript community recommends, popular JavaScript packages usually add and remove deprecated APIs on minor releases instead of removing them on major releases.

Table 5.4 – List of selected projects and number of versions.

Release Type	Number of Releases	Number of Increased Deprecated APIs	Number of Decreased Deprecated APIs
Major	127	57 (0.45/release)	19 (0.15/release)
Minor	1,357	1246 (0.92/release)	718 (0.53/release)

RQ2: Are deprecated APIs usually introduced and removed in major or minor releases? Popular JavaScript packages usually add and remove deprecated APIs on minor releases instead of removing them on major releases.

5.4 Threats to Validity

The study presented in this chapter has some limitations that could potentially threaten our results, as we explain next. First, we focused the historical analysis on 50 popular JavaScript open-source packages, according to Libraries.io. As a result of our decisions, our findings cannot be generalized to other systems implemented in other languages or closed-source packages. Additionally, we analyzed packages with a large number of dependent clients, as we expect them to be examples of well-maintained packages and representative case studies of open-source packages with many dependent clients. However, their maintainers may not represent the whole population of JavaScript developers. However, GitHub is the most popular software development platform, for both public and private packages. Future replication work on this topic could be conducted to address these issues.

Second, to identify deprecation occurrences, we only searched for matches of *deprecat*. Although being deliberate, this choice might have caused us to miss cases in which other terms are used. Furthermore, since we mine all JavaScript files, the deprecation strategies we analyzed might also be related to internal APIs that are not visible to consumers. Future studies that select only external APIs should address this issue. Moreover, the JavaScript tool for the mining study was implemented upon Flow, a well-known JavaScript code parsing library maintained by Facebook and, thus, the risk of errors is reduced. Additionally, we have manually inspected its output (each sample with a confidence level of 95% and a confidence interval of 5%). We find a precision of 98% for deprecation utility, 81% for code comment, 100% for JSDoc comment, and 100% for console message. Thus, our script can be used with a good level of confidence. Finally, the categorization of the deprecation occurrences we mined is subjected to the author/interpreter bias, although other members of our group verified the categories.

5.5 Final Remarks

In this chapter, we presented an historical analysis of API deprecation in popular JavaScript packages. This work can help the software engineering community better understand how JavaScript deprecated APIs are maintained over time.

We downloaded the top-50 popular JavaScript packages, according to Libraries.io, and analyzed their source code from different versions to analyze how deprecated APIs evolve. After investigating deprecation trends on those packages, our results suggest that most packages (59.4%) indicate increasing trends of deprecated APIs. When we look at specific deprecation strategies, we observe that 70% of packages with the deprecation utility type present upward trends. Additionally, we note that the deprecation mechanism with higher downward trends is console message, in which the usage has gone down in 29.4% of the analyzed packages. Furthermore, our results indicate that the number of deprecated APIs, in general, tend to increase at a higher rate than they decrease. We also investigated when deprecated APIs are usually introduced and removed. As a result, we observed that most deprecation occurrences are both added and removed on minor releases, contradicting what was recommended in the survey study in Chapter 3.

In the next chapter of this dissertation, we present the final considerations by concluding the dissertation, presenting the main contributions and limitations of this work and introducing insights for future work.

Chapter 6

6 Final Considerations

Understating JavaScript deprecation practices is important as insights can be provided about how developers are actually handling deprecated APIs. Additionally, it might also benefit developers in several ways, such as revealing common deprecation strategies in the ecosystem, endorse a clear communication environment in the community, improve the overall quality of JavaScript packages, and ease maintenance work. In this chapter, we present the final considerations regarding this dissertation. We first conclude our work by summarizing our motivation, goals, methodological procedures, results and contributions. Then, we discuss the main contributions of this dissertation. Finally, we give directions for future work.

6.1 Work Overview

JavaScript has become extremely popular over the last few years, and has been reported as the most commonly used programming language for several consecutive years. Furthermore, software reuse has become a key factor for a cost and time efficient software development package (Uddin et al., 2011). npm has reached over one million hosted JavaScript packages, making it the largest software repository to date (Tal and Maple, 2019). Despite the growth on the usage of JavaScript external libraries and APIs, little is known about JavaScript API deprecation mechanisms and practices. Additionally, there are no detailed studies related to this topic in the JavaScript ecosystem.

To fill these research gaps, we proposed three empirical studies. First, we conducted a survey study with developers to understand what API deprecation strategies are most commonly present in JavaScript packages and libraries. Results suggest that there is no standard or preferable strategy to deprecate JavaScript APIs. In general, the most common deprecation mechanisms are console message, project documentation, JSDoc annotation, and code comment. Additionally, developers learn about deprecated JavaScript APIs primarily via console message and project documentation. Also, most JavaScript developers (70%) only address deprecation issues if necessary or if time permits. Furthermore, we presented an extensive analysis on the current state of practice of JavaScript deprecation, along with approaches recommended by developers for deprecating APIs. In summary, developers suggested that planning a deprecation strategy and making clients aware of it is an efficient way

to handle deprecation in a JavaScript project. Furthermore, respondents advised to retain less deprecated APIs in favor of code health. However, if they need to be maintained, either add UNSAFE__ or similar prefix to a deprecated API or provide them through an opt-in flag, such as –legacy or –insecure.

In the second study, we proposed a mining study aiming at analyzing API deprecation mechanisms in popular JavaScript libraries. To achieve this goal, we downloaded the top 320 popular JavaScript packages on npm and analyzed their source code to identify and classify API deprecation occurrences. After analyzing those packages, results suggest that there is no standard approach to deprecate JavaScript APIs and there is no consistency in implementing a deprecation strategy. Still, we find that deprecation utility is the most frequent solution (41.7%), followed by code comment (34.5%), JSDoc annotation (18.8%) and console message (4.7%). Additionally, we find that those deprecation mechanisms might be used alone or combined at package and file levels. Over half of the analyzed packages (52.5%) adopt only one deprecation mechanism, while the remaining part combines two or more deprecation strategies. The most adopted single strategy is *code comment*, whereas the most common combination is *deprecation utility* and *code comment*.

Lastly, our third study investigates how deprecated APIs evolve over the lifetime of JavaScript packages. This time, we downloaded the top 50 popular JavaScript packages, according to Libraries.io, and analyzed their source code among 1918 different versions. After investigating deprecation trends on those packages, our results indicate that most packages (59.4%) present increasing trends of deprecated APIs. Looking at deprecation strategies separately, we note that 70% of packages with deprecation occurrences of the deprecation utility type present upward trends. Additionally, we note that the deprecation mechanism with higher downward trends is console message, in which the usage has gone down in 29.4% of the analyzed packages. Furthermore, our results suggest that the number of deprecated APIs, in general, tend to increase at a higher rate than they decrease. Additionally, we observed that most deprecation occurrences are usually both added and removed on minor releases rather than on major ones.

6.2 Contributions

We believe this dissertation has important contributions to the software engineering research community and industry. Next, we present our main contributions.

- We provide a novel large-scale study on JavaScript API deprecation practices and strategies adopted by developers and popular JavaScript packages;
- We present insights and thoughts regarding the current state of JavaScript depreciation provided by actual developers. We believe that information can contribute to future work;
- We provide a set of recommendations and good practices for deprecating APIs in JavaScript, also supplied by actual developers;
- We show the most common deprecation strategies adopted on popular JavaScript open-source packages, and how they are combined together. This can support other professionals during API design processes, and make consumer developers more aware of deprecation mechanisms;
- We present an overview of how deprecated APIs evolve over time in popular JavaScript packages, revealing historical trends and change rates. Additionally, we show when deprecated APIs are usually added and removed.

6.3 Future Work

As future work, we plan to go further and interview library maintainers to understand their rationale behind the adoption of multiple deprecation strategies or ad-hoc local solutions. Furthermore, our survey study brought to our attention the practice of introducing breaking changes communicated by other means, such as Semantic Versioning, project documentation and social media, in preference to API deprecation. We hypothesize that the fast-moving JavaScript community might prefer such approaches in favor of package publication speed. However, we wonder to what extent JavaScript developers are aware of Semantic Versioning to update project dependencies. That also remains a future work plan.

As future work, we plan to investigate other characteristics of API deprecation, such as replacement messages and their structure, external documentation and API evolution. We also plan to extend this research by creating a tool to automatically identify deprecation, suggest replacement messages, and alert developers about deprecated APIs. We plan to implement this tool and make it available for developers. Finally, based on our findings, we plan to propose guidelines on JavaScript API deprecation best practices that help and improve developers' experience.

BIBLIOGRAPHY

Bogart, C., Kästner, C., Herbsleb, J., & Thung, F. (2016). How to break an api: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 109-120).

Brito, A., Valente, M. T., Xavier, L., & Hora, A. (2020). You broke my code: Understanding the motivations for breaking changes in APIs. *Empirical Software Engineering*, 25(2), 1458-1492.

Brito, G., Hora, A., Valente, M. T., & Robbes, R. (2018). On the use of replacement messages in API deprecation: An empirical study. *Journal of Systems and Software*, 137, 306-321.

Granli, W., Burchell, J., Hammouda, I., & Knauss, E. (2017). The driving forces of API evolution. In *Proceedings of the 14th International Workshop on Principles of Software Evolutio*n (pp. 28-37).

Hussain, M. & Mahmud, I. (2019). pymannkendall: a python package for non-parametric Mann Kendall family of trend tests. *Journal of Open Source Software*, 4(39), 1556.

Li, L., Gao, J., Bissyandé, T. F., Ma, L., Xia, X., & Klein, J. (2018). Characterising deprecated android apis. In *International Conference on Mining Software Repositories* (pp. 254-264).

Moser, S. & Nierstrasz, O. (1996). The effect of object-oriented frameworks on developer productivity. *Computer*, 29(9), 45-51.

Robbes, R., Lungu, M., & Röthlisberger, D. (2012). How do developers react to api deprecation? The case of a Smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (pp. 1-11).

Sawant, A. A., Robbes, R., & Bacchelli, A. (2016). On the reaction to deprecation of 25,357 clients of 4+1 popular java APIs. In *2016 IEEE International Conference on Software Maintenance and Evolution* (ICSME) (pp. 400-410).

Sawant, A. A., Aniche, M., van Deursen, A., & Bacchelli, A. (2018a). Understanding developers' needs on deprecation as a language feature. In *2018 IEEE/ACM 40th International Conference on Software Engineering* (ICSE) (pp. 561-571).

Sawant, A. A., Huang, G., Vilen, G., Stojkovski, S., & Bacchelli, A. (2018b). Why are features deprecated? An investigation into the motivation behind deprecation. In *2018 IEEE International Conference on Software Maintenance and Evolution* (ICSME) (pp. 13-24).

Sawant, A. A., Robbes, R., & Bacchelli, A. (2018c). On the reaction to deprecation of clients of 4 + 1 popular java apis and the jdk. *Empirical Software Engineering*, 23(4), 2158-2197.

Sawant, A. A., Robbes, R., & Bacchelli, A. (2019). To react, or not to react: Patterns of reaction to API deprecation. *Empirical Software Engineering*, 24(6), 3824-3870.

Tal, L. & Maple, S. (2019). npm passes the 1 millionth package milestone! what can we learn? In *https://snyk.io/blog/npm-passes-the-1-millionth package-milestone-what-can-we-learn*. Last access: Nov, 2019.

Tourwe, T. & Mens, T. (2003). Automated support for framework-based software. In *International Conference on Software Maintenance,* 2003. ICSM 2003. Proceedings. (pp. 148-157).

Uddin, G., Dagenais, B., & Robillard, M. P. (2011). Analyzing temporal API usage patterns. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering* (ASE 2011) (pp. 456-459).

Wang, J., Li, L., Liu, K., & Cai, H. (2020). Exploring how deprecated python library apis are (not) handled. In *Proceedings of the 28th acm joint meeting on European Software Engineering Conference and Symposium on The Foundations of Software Engineering* (pp. 233-244).

Xavier, L., Brito, A., Hora, A., & Valente, M. T. (2017). Historical and impact analysis of API breaking changes: A large scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering* (SANER) (pp. 138-147).

Yasmin, J., Tian, Y., & Yang, J. (2020). A first look at the deprecation of restful APIs: An empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 151-161).