On the (Un-)Adoption of JavaScript Front-end Frameworks

Fabio Ferreira[1,2], Hudson Silva Borges[3], and Marco Tulio Valente[1]

[1]Department of Computer Science, UFMG, Brazil
[2]Center of Informatics, IF Sudeste MG - Campus Barbacena, Brazil
[3]Department of Computer Science, UFMS, Brazil

Abstract

JavaScript is characterized by a rich ecosystem of libraries and frameworks. A key element in this ecosystem are frameworks used for implementing the front-end of web-based applications, such as Vue and React. However, despite their relevance, we have few works investigating the factors that drive the adoption—and un-adoption—of front-end-based JavaScript frameworks. Therefore, in this paper, we first report the results of a survey with 49 developers where we asked them to describe the factors they consider when selecting a front-end framework. In the second part of the work, we focus on projects that migrate from one framework to another since JavaScript's ecosystem is also very dynamic. Finally, we provide a quantitative characterization of the migration effort and reveal the main barriers faced by the developers during this effort. Although not completely generalizable, our central findings are as follows: (a) popularity and learnability are the key factors that motivate the choice of front-end frameworks in JavaScript; (b) from the 49 surveyed developers, one out of four have plans to migrate to another framework in the future; (c) the time spent performing the migration is greater than or equal to the time spent using the old framework in all studied projects. We conclude with a list of implications for practitioners, framework developers, tool builders, and researchers.

Keywords— JavaScript , Front-end frameworks, software modernization, software reengineering.

1 Introduction

JavaScript is the facto programming language for the Web, which explains its relevance and continuous growth. For example, since 2014, the language is the most popular one on GitHub, according to GitHub's Octoverse report.[1] JavaScript is also characterized by a large, rich, and dynamic ecosystem of frameworks and libraries [1]. For example, npm—the largest package repository for the language—hosts more than 1.4 million projects.[2]

[1]https://octoverse.github.com/
[2]https://www.npmjs.com/

In the JavaScript ecosystem, a key class of systems are frameworks for building the front-end of Web-based systems, such as Vue, React, Angular, Ember, and Backbone. These frameworks provide an architecture and components for building modern, rich, and responsible Web interfaces, which are usually called *Single-Page Applications* (SPAs) [2, 3]. Their importance can be illustrated by the number of stars the respective projects have on GitHub. For example, at the time of this writing, both Vue and React have more than 150K stars. Furthermore, these frameworks are usually developed and maintained by major Internet companies, as is the case of React (Facebook) and Angular (Google).

However, despite these numbers and supporters, we have a limited list of papers that study the relevance and adoption of front-end frameworks in the JavaScript ecosystem. An important exception is a study by Pano, Graziotin, and Abrahamsson [4], where the authors interviewed 18 decision-maker developers regarding their criteria for selecting JavaScript frameworks, and as a result, they propose a model of framework adoption factors. However, their study was conducted in July 2014, when front-end frameworks were still in the early adoption phases. For example, the most popular framework nowadays (Vue) was not mentioned by any participant and a single participant mentioned the second most popular framework nowadays (Facebook's React). By contrast, jQuery—which is not widely used anymore—was cited by half of the interviewed developers. Furthermore, the authors investigated only the factors driving the adoption, but not the un-adoption of a framework. In dynamic software ecosystems, frameworks' un-adoption is a worth studying effort.

Therefore, in this paper, we study both the adoption and the un-adoption of JavaScript frameworks. Specifically, in the un-adoption study, we focused on frameworks' migrations, which consists of un-adopting one framework and then replacing it by another one (since in practical terms, it is very difficult to remove a front-end framework dependency and do not replace it by another framework). To this purpose, we first select five front-end frameworks, including the two most popular ones (Vue and React), one leading framework in the past (Angular) and two less adopted ones (Ember and Backbone). This diversity is important to increase the chances of capturing the two events of interest in this paper, i.e., framework's adoption and un-adoption. Next, we curate two datasets: one dataset with 1,515 popular projects that are clients of the studied frameworks and a second dataset with 19 projects that successfully completed a migration in the past.

We use the first dataset to qualitatively study the factors driving the adoption of contemporary front-end JavaScript frameworks. To this purpose, we performed a survey with 49 core front-end developers of such projects, asking them about the central reasons for using the studied frameworks in their projects. We also asked the participants about possible plans to adopt a distinct front-end framework in the future.

The second dataset is used to study framework's un-adoption. We use it to answer three research questions:

RQ1: How much time was spent to conclude the migration from one front-end framework to another?

RQ2: What was the effort spent in these migrations?

RQ3: What were the version control practices used to perform the migrations?

Among the version control practices referred to in the third research question, we investigate the use of migration-specific branches, forks and clones, and migrations conducted directly on the master or development branch.

Finally, we conducted a survey with 11 developers directly involved in the studied framework migrations. We used this final survey to clarify three key aspects of the migrations: (1) the main reasons that motivated the decision to migrate from one framework to another; (2) the main barriers faced in this process; (3) the main challenges and benefits of using a specific repository practice during the migrations.

Contributions. Our contributions are threefold:

- We reveal a list of nine key factors developers consider when selecting contemporary JavaScript front-end frameworks. This list is useful to JavaScript developers that plan to adopt a front-end framework in their projects. It can also be used by framework developers, helping them better position their projects in a very competitive software market.

- We show there is a real risk of framework's migration in JavaScript projects. Out of 49 surveyed developers, we found that five have concrete plans to migrate their projects to another framework, seven mentioned they plan to use a different framework in future projects, and six developers have already completed a migration in private repositories. These numbers might motivate tool builders and researchers to investigate, design, and implement tools to help with this process.

- Finally, we report the effort spent, the employed version control practices, and the barriers faced by projects that successfully concluded a framework migration. This information is useful to practitioners that plan to start a similar effort in their projects.

Structure. This paper is structured as follows (see also a visual representation of the paper´s key sections in Figure 1). Section 2 describes the JavaScript frameworks studied in this paper. Section 3 describes the two datasets used in the study: the first dataset includes JavaScript projects whose developers were surveyed about their motivation to adopt a JavaScript front-end framework; the second dataset has projects that successfully completed a framework migration in the past. Section 4 reports the results of the survey conducted to reveal the major factors driving the adoption of JavaScript front-end frameworks. In Section 5, we focus on the migration of front-end frameworks, studying this event under a quantitative and qualitative lens. The implications and lessons learned are discussed in Section 6. Sections 7 and 8 describe threats to validity and related work, respectively. Section 9 concludes and presents ideas for future work.

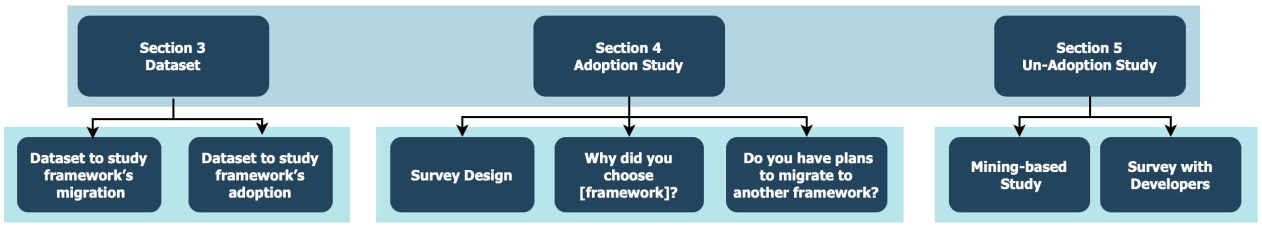

Section 3
Dataset
Section 4
Adoption Study
Section 5
Un-Adoption Study
Dataset to study framework's migration
Dataset to study framework's adoption
Survey Design
Why did you choose [framework]?
Do you have plans to migrate to another framework?
Mining-based Study
Survey with Developers

Figure 1: Key sections of the paper

2 JavaScript Front-end Frameworks

This section briefly describes the five JavaScript frameworks studied in this paper: Angular,[3] React,[4] Vue,[5] Ember,[6] and Backbone.[7] They were selected due to their popularity (nowadays and in the past) and importance to modern web software development. However, since we focus this study on adoption and un-adoption events, we did not select relevant frameworks released very recently, such as Preact and Svelte, because they are not good candidates for the un-adoption study.

The frameworks considered in this study have a lot in common. They are open-source, allow the creation of reusable UI components, and provide an architecture for creating Single-Page Applications (SPAs). In addition, they allow writing HTML and JavaScript code together. However, they differ in some aspects, for example, React allows writing HTML in JavaScript, while Angular and Vue allow writing JavaScript in HTML. Next, we briefly comment on each one.

Angular was released by Google in 2010. In 2016, Google launched a new version of the system, which was rewritten from scratch. Angular dictates a MVC (Model-View-Controller) architecture [5]. Furthermore, a two-way data binding mechanism is employed to keep the view and model in sync. This mechanism automatically reflects model changes in the view and vice-versa.

React—released by Facebook in 2013—does not enforce any architectural pattern, such as MVC. Instead, the framework prioritizes interoperability, i.e., it can be incorporated on existing systems and allows gradual adoption. React supports one-way data binding, when changes in the model are automatically propagated to the view, but not in the other way.

Vue is a progressive framework released by Evan You, in 2014. It is based on a so called Model–View–ViewModel (MVVM) pattern that separates an application into Model (data logic), View (GUIs), and ViewModel (directives). Vue supports both one-way and two-way data binding.

Ember was created in 2011 by Yehuda Katz. It provides templates, components, models, and routes. It also supports two-way data binding.

[3] https://github.com/angular/angular
[4] https://github.com/facebook/react
[5] https://github.com/vuejs/vue
[6] https://github.com/emberjs/ember.js
[7] https://github.com/jashkenas/backbone

Backbone was released by Jeremy Ashkenas in 2010 and therefore is the oldest framework we will study. Similar to React, Backbone does not enforce any architectural pattern and supports only one-way data binding.

3 Dataset

As the first step for creating a dataset with clients of our target frameworks, we retrieved the names that identify them in two popular package managers: npm and Bower. For this purpose, we randomly selected 10 GitHub projects that are clients of each framework (using GitHub's Used-by feature) and inspected their *package.json* and *bower.json* files. Table 1 describes the package names we found after this step.

Table 1: Package names used by the studied frameworks

Framework	**Package Names**
Vue	vue
React	react
Angular	angular, @angular/core
Backbone	backbone
Ember	ember, ember-cli, ember-cli-app-version

Next, we selected the top-15,000 most popular projects on GitHub ranked by stars, which is a metric commonly used to rank projects by popularity [6, 7]. Then, we discarded 541 projects labeled as archived by GitHub. We also searched for forks, but we did not find anyone among the selected projects. Finally, for the remaining 14,459 projects ($15,000 - 541$), we checked out all versions of their *package.json* and *bower.json* files in order to retrieve the project's dependencies. If a project has never depended on any of the five frameworks of interest, it was discarded.

We found 1,515 projects that are (or were) clients of the studied frameworks, since they have (or had) a dependency to one of them. Table 2 shows the number of clients by framework. As can be observed, React is the most popular framework (988 clients, 65.2%), followed by Vue (315 projects, 20.7%), and Angular (263 clients, 17.3%). By contrast, Ember has only 11 clients (0.7%).

Table 2: Number of clients by framework

Framework	**# Clients**
React	988
Vue	315
Angular	263
Backbone	68
Ember	11

Table 3: Dataset to study framework's migration

From	**To**	**# Projects**
Angular	React	8
Vue	React	6
Angular	Vue	3
Backbone	React	1
React	Vue	1

3.1 Dataset to study framework's migration

From the 1,515 selected client projects, 116 projects have at least two dependencies to the studied frameworks in their commit history. However, we discarded 50 projects that do not represent a practical case of framework migration, including 38 projects that only use the studied frameworks in examples, documentation or similar files, seven projects that only use the frameworks in test code, three non-software projects, one deprecated project, and one project that is a copy of another repository without using a fork.

We manually inspected the remaining 66 projects ($116 - 50$) to understand better why they are using (or have used) the frameworks. Essentially, we analyzed several artifacts (e.g., commits, issues, and pull requests) to find formal evidence of the maintainers' intention to move from a framework X to framework Y. Unfortunately, in the case of 40 projects, we did not find such artifacts. Thus, we discarded them to focus on projects with a well-documented migration initiative.

The result list consists of 26 projects ($66 - 40$), which we categorized into three major groups (assuming that X and Y are one of the frameworks used in this paper):

- *Migration completed*: 19 projects that replaced a dependency to X by a dependency to Y.
- *Ongoing migration*: five projects that, at the time of the dataset collection, depend on both X and Y.
- *Migration interrupted*: two projects that replaced a dependency to X by a dependency to Y. However, they reverted to use X later.

Since they are more common, we focused the un-adoption study (Section 5) on projects that completed a framework migration (19 projects). Table 3 shows the origin (X) and target framework (Y) for such projects. As we can see, the results show that JavaScript developers moved to React (15 projects) after using Angular (8 projects), Vue (6 projects) or Backbone (1 project). Our dataset also includes three projects that migrated from Angular to Vue and one project that migrated from React to Vue.

Example: As an example, we have project Redocly/redoc. In August 2017, the project's maintainers opened an issue to discuss a migration from Angular to React (Figure 2).[8] Two

[8] https://github.com/Redocly/redoc/issues/327

months later (October 2017), the first dependency to React was introduced in the project's *package.json* file, i.e., the migration started. In January 2018, the last dependency to Angular was removed from the project, i.e., the migration finished.

Hey, ReDoc community!

To support OpenAPI 3.0 I am working on a major refactor of the codebase. As part of this refactor I am considering rewriting ReDoc view layer completely to React. Why? I've been working with React on a few side project for the last 5-6 month and I really loved working with it comparing to Angular:

** much much simpler, no magic*
** more performant (especially since React 16 release aka Fiber)*
** the community is healthier - much more stable ready-to use modules (there are lots in Angular too, but usually they are part of some huge UI-packs)*
** smaller bundle size*
** simpler to build-in live updates*
** server-side rendering actually working*
** I love styled-components*

Figure 2: Example of issue documenting a framework migration

3.2 Dataset to study framework's adoption

We defined a second dataset with projects that did not face a migration, i.e., projects that have not been considered in the first dataset. Initially, we defined that this second dataset should have at most 10% of the clients initially collected for each framework (or ten clients, selecting the greater value). The reason is that we rely on this dataset to conduct a survey with developers and we do not intend to send a massive number of e-mails in order to avoid our research being perceived as spam [8].

After randomly selecting the first sample of 10% of the clients, the first author of this paper carefully analyzed each one to discard non-software projects, archived projects, and projects that use the frameworks only in examples, tutorials, documentation, and tests. After that, new projects were randomly selected to replace the discarded ones. This procedure was repeated until we reached the target number of clients for each framework or had no more projects to select. As a result, we selected a sample of 169 projects, as presented in Table 4. React has the highest number of clients (99 projects), followed by Vue (32 projects). By contrast, we found only one eligible Ember project.

4 Adoption Study

To study the factors that motivate the adoption of front-end frameworks, we surveyed developers of our second dataset.

Table 4: Dataset to study framework's adoption

Framework	**# Projects**
React	99
Vue	32
Angular	27
Backbone	10
Ember	1

4.1 Survey Design

In this survey, we used the dataset of projects that did not face a migration, as detailed in Section 3.2. To increase the chances of receiving accurate responses, we decided to send the survey only to the project's core front-end developers. To identify such developers, we used a Commit-Based Heuristic but taking into account only commits that change front-end-related source code files, according to the extensions listed in Table 5. In other words, we adapted the heuristic commonly used in the literature to identify core developers to the context of front-end files [9, 10]. After this adaptation, core front-end developers are the ones responsible for at least 80% of the commits performed in front-end related files.

Table 5: Front-end files extensions

`*.js, *.jsx, *.ts, *.tsx, *.es, *.es6, *.mjs, *.vue`
`*.htm, *.html, *.xhtm, *.xhtml, *.css, *.scss, *.sass`

Next, we retrieved the email addresses of the core front-end developers from their commits' metadata. We sent only one email per project, addressed to its top core front-end developer, i.e., the one with the highest number of commits. We also sent at most one email to a given developer, i.e., whenever we selected a core front-end developer who was emailed before—for another project—, we selected the next one. Again, our intention was to avoid developers perceiving our messages as spam.

In the email, we asked only two questions:

1. *Why did you choose [framework]?*

2. *Do you have plans to migrate to another framework? Please explain.*

With the first question, our goal is to reveal the factors that drive the adoption of JavaScript front-end frameworks. With the second question, we intend to unveil possible intentions to adopt another framework in the future.

We sent 169 emails and received 49 responses, achieving a response rate of 29%. After collecting all responses, we analyzed the answers using thematic analysis, a technique for identifying and recording patterns (or "themes") within a collection of documents [11, 12]. The analysis involves the following steps: (1) initial reading of the developer responses; (2) generating initial codes for each response; (3) searching for themes among codes; (4) reviewing the themes to find opportunities for merging; and (5) defining and naming the final themes. The first author of this paper performed these five steps. Two other authors then reviewed the final classification. When quoting the answers, we use labels RA1 to RA49 to indicate the respondents.

4.2 Why did you choose [framework]?

We identified nine key factors that motivate the adoption of front-end frameworks in JavaScript. Table 6 summarizes these factors and provides a brief description of each one. Some respondents mentioned more than one factor. Thus, the number of occurrences in Table 6 is greater than 49.

Table 6: Factors motivating the adoption of front-end frameworks

Factor	**Description**	**Occurences**
Popularity	This framework is widely known and used	19
Learnability	This framework is easy to learn and use	17
Architecture	This framework forces clients to follow a solid architecture	15
Expertise	I have previous expertise in this framework	10
Community	This framework is maintained by a large community	9
Performance	This framework has excellent performance	8
To gain experience	I wanted to gain experience in this framework	6
Documentation	This framework's documentation is very good	5
Sponsorship	This framework is supported by well-known companies	4

To better understand our results, we also divided the factors by framework, as presented in Table 7. Then, in the following paragraphs, we describe and give examples of the top factors that influence the adoption of each framework.

REACT*:* We received 20 answers from REACT's developers. Popularity (8 answers), architecture (7 answers), and community (6 answers) are the most common adoption factors cited by them. As examples, we have the following answers:

I chose React because it was emerging as the most popular UI. Framework popularity doesn't always mean the best, but it does mean you have a larger talent pool to draw from, a larger pool of supporting libraries in the ecosystem, more tutorials, etc. (RA26 - Popularity)

React allowed us to have a constant velocity whatever the size of the application, thanks to an architecture based on composition and the one-way data binding encouraged by Flux. (RA35 - Architecture)

I/we are very happy with the stability and community that React offers (no major bugs we've seen

Table 7: Factors that motivate the adoption of JavaScript front-end frameworks (answers per individual frameworks).

Factor	**Total**	**Ember**	**Backbone**	**Angular**	**Vue**	**React**
Popularity	19	0	2	5	4	8
Learnability	17	0	1	0	11	5
Architecture	15	0	1	3	3	8
Expertise	10	0	1	2	4	3
Community	9	0	0	1	2	6
Performance	8	1	1	1	5	1
To gain experience	6	0	0	2	0	4
Documentation	5	0	1	0	3	1
Sponsorship	4	0	0	1	0	3

and people are creating and maintaining libraries for all sorts of things). (RA44 - Community)

VUE*:* The framework stands out for its simplicity and ease of use. 11 (out of 14 respondents, or 78.6%) decided to use VUE because it is easy to learn. Other factors are performance with five answers, popularity, and previous expertise, each with four answers. As examples, we have the following responses:

Because Vue.js is much easier to start working with, one can easily transfer the team from existing technology to Vue.js. We've seen guys without previous Vue.js experience, building complete e-shops in 2–3 weeks. (RA17 - Learnability)

We chose Vue specifically because it felt simpler, easier to learn, and we already had someone in our team experienced with Vue. (RA18 - Learnability and Expertise)

Vue because the documentation is great, it is easy to learn and it is really performant. (RA24 - Documentation, Learnability and Performance)

Another reason to go for VueJS was its popularity, so developers can contribute more easily. (RA23 - Popularity)

ANGULAR*:* Popularity—not nowadays, but at the time of the adoption— was cited by five ANGULAR's respondents (50%), as in the following quote:

At the time Angular 1 was the hottest framework in town. This was before React, before Vue. (RA5 - Popularity)

BACKBONE*:* Similar to ANGULAR, developers adopted BACKBONE due to its popularity at the time:

It was the most popular library at the time, and I had a lot of experience with it. (RA2 - Popularity *and Expertise)*

Ember*:* We received the following answer about Ember:

Ember was a fully featured framework, but If I had the option, I would probably have chosen React at the time or Angular if we were to build it now. (RA1 - Ember*)*

Summary: Popularity (39%) and learnability (35%) are the main factors that motivate the adoption of front-end frameworks in JavaScript.

Weakness Factors: Beyond the factors that motivate adopting a JavaScript front-end framework, some respondents mentioned why they did not choose other frameworks. After carefully analyzing such answers, we identified eight factors considered as weaknesses by the respondents. Table 8 summarizes these factors for each framework.

Table 8: Weakness factors of JavaScript front-end frameworks (answers per individual frameworks).

Factor	**Total**	**Ember**	**Backbone**	**Angular**	**Vue**	**React**
Incompatibility of versions	7	0	0	7	0	0
Complexity	4	0	0	4	0	0
Suboptimal architecture	4	0	1	2	0	1
Lack of experts	4	0	1	0	0	3
Difficult to maintain	3	0	1	1	0	1
Lack of TypeScript Support	3	0	1	0	2	0
Poor performance	1	0	0	1	0	0
Security issues	1	0	0	1	0	0

In the following paragraphs, we briefly describe and give examples of the weak points of each framework.

React: Three developers cited the lack of experts as the key reason for not choosing React as their front-end framework. A clear example is the following answer:

The talent pool for experienced React devs is limited. (RA26)

Vue: At the time of the survey, two developers cited the lack of support to TypeScript as one of the Vue's drawbacks. However, this issue is solved in Vue 3, as also mentioned by the participant:

If the project grows much larger than current implementation in the future, Vue 3.0 might be adopted as it better supports TypeScript. (RA18)

Angular: The incompatibility between Angular 1 and Angular 2 is a key factor for developers not choosing Angular. Seven respondents mentioned this problem in their answers:

We were using AngularJS version 1, and when version 2 came out, with a completely new syntax that would have required rewriting most of the code, we decided to see if there were other options. (RA20)

Backbone: Three developers commented on Backbone's drawbacks. The difficulty to find developers with expertise in Backbone (1 answer), the framework's suboptimal architecture (1 answer), code maintainability problems (1 answer), and the lack of support to TypeScript were pointed as drawbacks of the framework.

Ember: Only one developer comment about Ember:

I chose React for [my-project] in 2016 because it was emerging as the most popular UI framework. At the time, it was clear Ember would not reach the popularity that React and Angular were reaching. (RA26)

4.3 Do you have plans to migrate to another framework?

We received 46 answers for this question. Five (10.9%) out of 46 respondents explicitly expressed the intention to migrate to another framework: one respondent from Ember to Angular, one from Backbone to React, two from Angular to React, and one from Angular to Vue. The main reasons are difficulty in hiring developers and difficulty to maintain the codebase, as in this answer:

Yes, we already migrated our paid products to React and are going to update the open source codebase as well. The main reason for migration was (1) the simplicity of React and much more possibilities to create reusable component libraries; (2) It was hard to hire an engineer that knows Backbone; (3) It was a pain to maintain a huge codebase written using Backbone. (RA3)

Furthermore, 6 (13.0%) out of 46 respondents stated they already completed a migration before (one from Angular to Backbone, two from jQuery to Angular, two from Angular to Vue, and one from Backbone to Angular and next to React). As example, we received this answer:

We were using AngularJS version 1, and when version 2 came out, with a completely new syntax that would have required rewriting most of the code, we decided to migrate to Vue. (RA20)

These six projects were not included in our dataset of projects facing migration due to three reasons: two projects migrated from jQuery, which is not a framework for building single-page applications; three projects deleted the *package.json* file used to declare the dependency to the old framework;[9] and one project performed the migration in a new repository and turned the first one into bug-fixing only mode.

[9]Indeed, the file was deleted and another *package.json* was created in another folder. For this reason, the first file was not detected by the mining approach described in Section 3.1.

On the other hand, 35 respondents (76.1%) did not express intention to migrate to another framework. The main reasons are as follows: our system is working well (16 respondents), our project is in maintenance mode (3 respondents), and migrating to another framework requires a huge effort (4 answers). As examples, we have the following answers:

It would be nice to move to something TypeScript based, however as [my-project] is in maintenance only, there are no plans, only hopes. (RA2)

Not at all. The main goal is still to offer the best tool that solves a problem, and the roadmap is huge. Users do not really care which technology is used as long as it solves their problems and saves them some time. (RA6)

Rewriting it in either framework represents a huge work that nobody is willing to take at the moment. (RA10)

However, 7 (20%) out of 35 respondents that did not express the intention to migrate to another framework stated they would not use the same framework in a new project. As example, we have the following quote:

No plans to migrate [my-project]. However, for new projects that I have picked up since then and the ones I will pick up, my default choice is now Vue. The overall development experience of Vue feels way better to me and easier to wrap my head around than React. (RA33)

Interestingly, one respondent mentioned PReact, and one mentioned Svelte as an alternative in the future, for replacing React. We did not include these frameworks in our study because they are very recent, and thus they are not good candidates for the un-adoption study. For example, the respondent that commented on Svelte also indicated that five years are needed to make a decision:

In 5 years, if Svelte becomes extremely popular and the React ecosystem begins to shrink or become deprecated, we plan to consider a migration. (RA26)

Summary: 23.9% of the participants finished or pretend to start a migration to another framework. Moreover, 20% have plans to adopt a distinct framework in future projects.

5 Un-adoption Study

We divided our study of un-adoption into two parts. First, we analyzed the repositories of 19 projects that successfully completed a migration, using the dataset described in Section 3.1. Our intention was to answer three research questions: how much time was spent on the migrations (*RQ1*), how much effort was necessary, in terms of the number of developers and code churn (*RQ2*), and the version control practices used to deal with the migration (*RQ3*). Then, in the second part of the un-adoption study, we surveyed the key developers involved in the migration, to clarify some points investigated in the first study and also to shed light on other aspects of the un-adoption.

5.1 Mining-based Study

(RQ1) How long did the migrations take?

Assuming a migration from framework X to framework Y, we computed the time spent to complete the migration by subtracting the date when the dependency to X was removed from the *package.json* file, i.e., when the migration was completed, from the date when the dependency to Y was introduced in the project for the first time, i.e., when the migration to framework Y started. However, for six projects, the removal of the dependency to X and the addition of the dependency to Y happened in the same day, suggesting they used a private repository for migrating. For this reason, we did not consider these projects in RQ1 and RQ2.

To better represent the time spent on the studied migrations, we also divided each project's lifetime into three segments: time spent using the old framework X, time spent performing the migration, and time spent using the new framework Y. Figure 3a shows the percentage each time frame represents in the lifetime of the analyzed projects. As we can see, the time spent performing the migration is greater than or equal to the time spent using the old framework in all studied repositories. Moreover, 5 (41.6%) out of 12 repositories spent more time performing the migration than using the old and new frameworks.

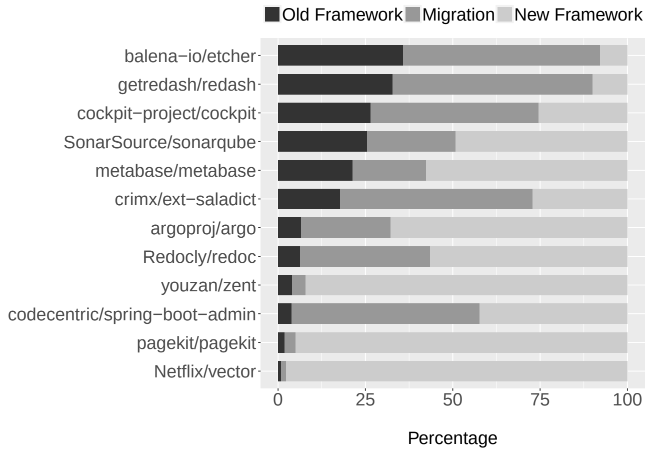
Figure 3b shows the number of days spent on the migrations. The number ranges from seven days (NETFLIX/VECTOR) to 966 days (BALENA-IO/ETCHER). The results confirm an expected property of frameworks migration: the migration takes less time when performed in earlier stages of a project lifetime. For example, the projects that spent less than three months on the migrations—VECTOR, ARGO, PAGEKIT, ZENT, and SPRING-BOOT-ADMIN— started the effort when they have used the old framework for at most 6.5% of their lifetime. By contrast, ETCHER and REDASH were the projects that spent more time migrating from frameworks. The main reason pointed by the maintainers was due to a monolithic architecture and the need to keep the application evolving, respectively.

Summary: Front-end framework's migration takes time. 5 out of 12 repositories spent more time performing the migration than using the old and new frameworks.

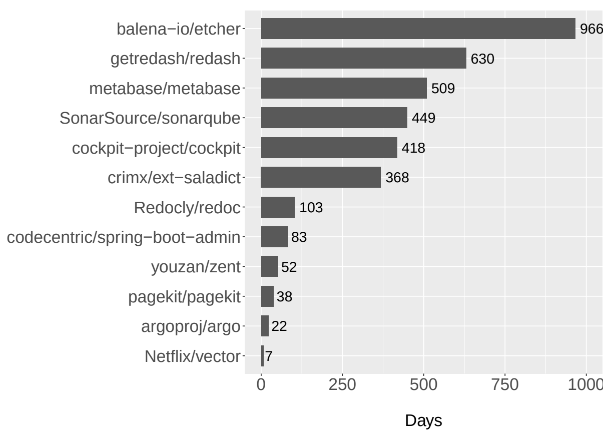
(RQ2) What was the effort spent on the migrations?

We initially used code churn—defined as the number of lines of source code added or deleted in a commit—as a proxy for the effort spent in the migrations [13]. However, we only compute code churn for source code files related to the front-end layer. We again assume that front-end-related files have one of the extensions in Table 5. We also only consider code churn for the commits performed in the migration time frames.

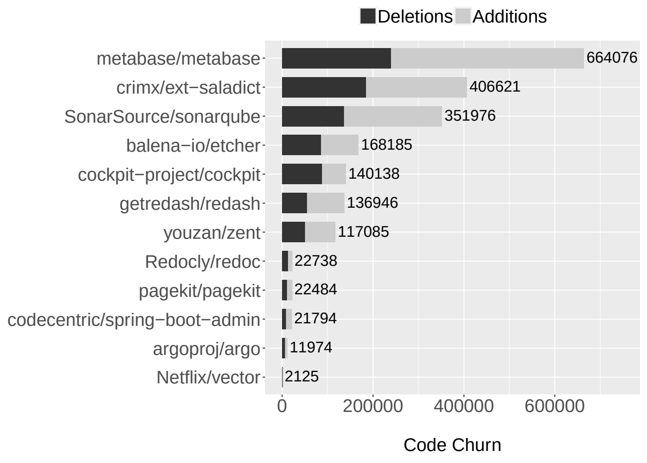
Figure 3c shows the code churn per project. The five projects with the lowest measures are exactly the ones that spent the lowest number of days to complete the migration. However, as code churn increases, other factors seem to influence the migration effort. For example, METABASE/METABASE has the highest code churn, but it did not demanded the longest time to



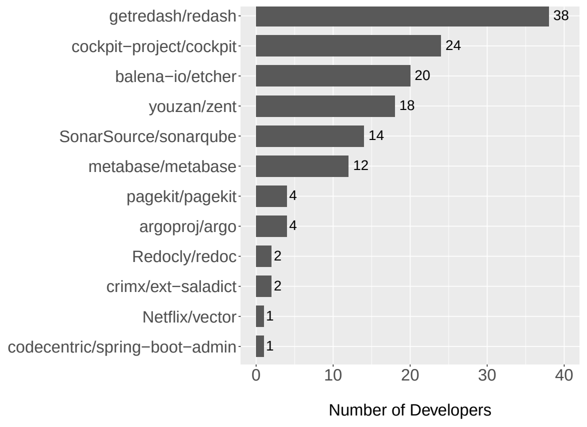
(a) Percentage of time before, during, and after the migration



(b) Duration



(c) Code Churn



(d) Number of Developers

Figure 3: Quantitative characterization of the studied migrations

complete the migration.

We also counted the number of developers responsible for the commits in the code churn analysis, as presented in Figure 3d. As we can see, the number ranges from one developer (Netflix/vector and codecentric/spring-boot-admin) to 38 developers (getredash/redash).

Summary: There is a wide variation in the number of developers working in front-end-related files during the migrations (from 1 to 38 developers).

Comparing the maintenance effort during and after the migration: In order to compare the migration effort with a "normal" maintenance effort, we also analyzed the changes in the front-end files after the migration. More specifically, suppose the migration lasted n months and changed a set of front-end files F. In this case, we also monitored the files in F for another n months after the migration to check whether they continued to be maintained.

Figure 4 shows the results in percentage terms. As we can see, in eight out of 11 systems, more than 50% of the migrated files were not changed at all after the migration. Thus, this result confirms the migration was indeed a relevant event in the history of such front-end files.

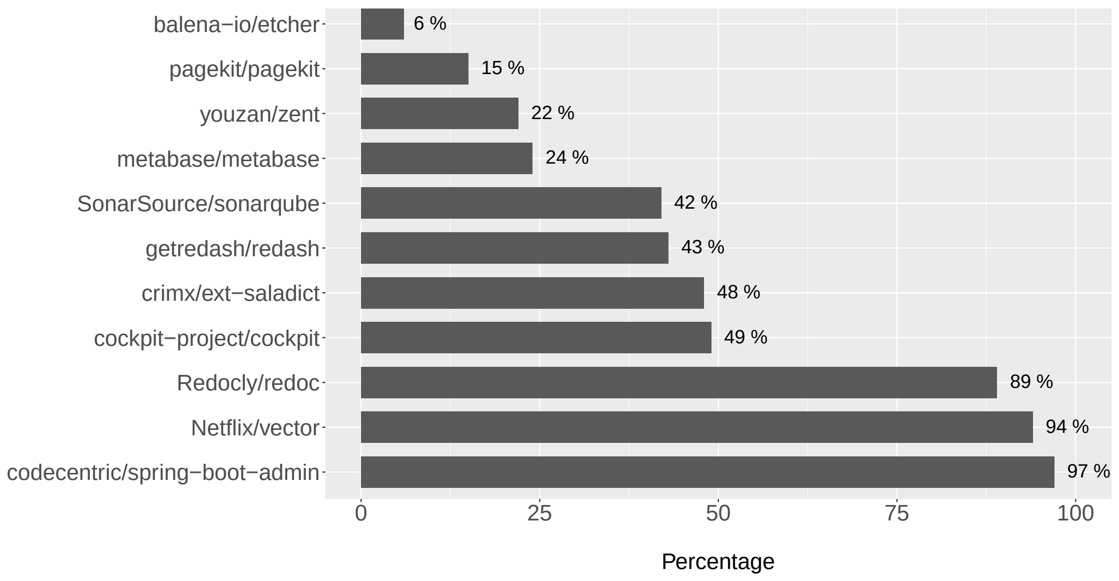


Figure 4: Percentage of front-end files changed after the migration

(RQ3) What were the version control practices used for migration?

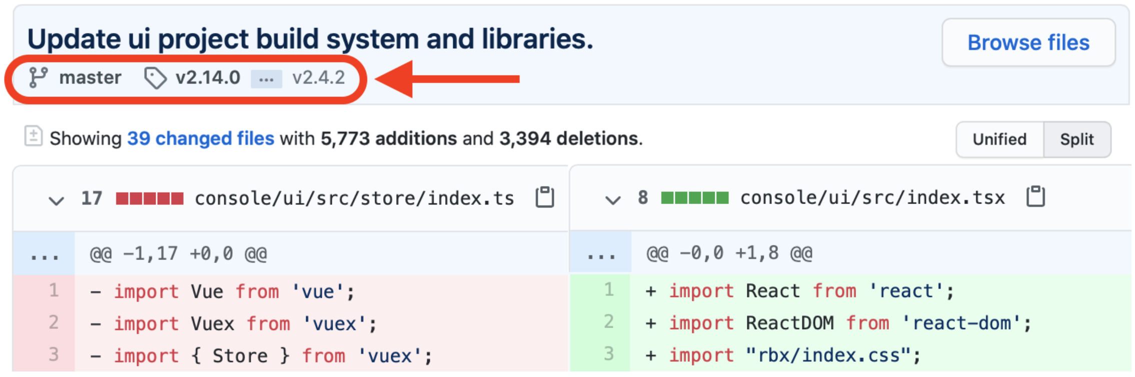
Initially, the first author searched for commits, pull requests, issues, etc., documenting the migrations and carefully inspected them for information on practices used to support two frameworks in the source code during the migration, such as branches or forks. Next, the second and third authors also analyzed each document to validate its content. In the remainder of this section, we detail each practice followed by the studied projects.

Master/Development branch (9 projects): these projects use the master or development branch during the migration, committing every modification directly to the main branch. Among these projects, four performed the migration in the same day (as we highlighted in Section 5.1). For example, Figure 5 shows a migration commit from HEROICLABS/NAKAMA which adds new features implemented using REACT and removes old VUE features.

Migration Branches (4 projects): these projects use branches explicitly created for the migration and usually delete them after merging on the master branch. Projects can use a branch for each feature under migration or a single branch throughout the migration process. For example, Figure 6 shows the REDOCLY/REDOC project, which created the *react-rewrite*[10] branch for supporting the migration.

Other branches (5 projects): We also have five projects where the migration happened in specific branches, but not explicitly created for migration purposes.

[10]https://github.com/Redocly/redoc/pull/357


Update ui project build system and libraries.

master v2.14.0 ... v2.4.2

Browse files

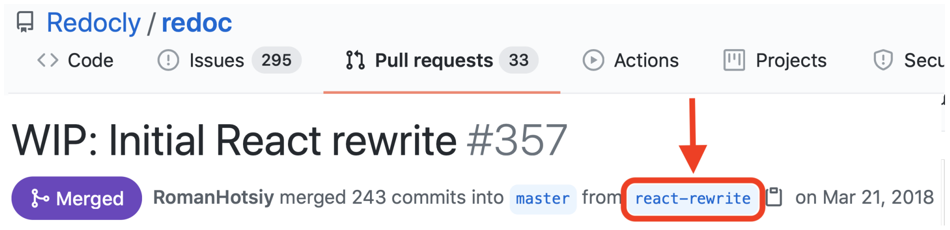
Showing 39 changed files with 5,773 additions and 3,394 deletions.

Unified Split

```
17 console/ui/src/store/index.ts
@@ -1,17 +0,0 @@
- import Vue from 'vue';
- import Vuex from 'vuex';
- import { Store } from 'vuex';
```

```
8 console/ui/src/index.tsx
@@ -0,0 +1,8 @@
+ import React from 'react';
+ import ReactDOM from 'react-dom';
+ import "rbx/index.css";
```

Figure 5: Commit performed in the master removing Vue code (left) and adding new code in React (right)


Redocly / redoc

Code Issues 295 Pull requests 33 Actions Projects Secu

WIP: Initial React rewrite #357

Merged RomanHotsiy merged 243 commits into `master` from `react-rewrite` on Mar 21, 2018

Figure 6: Merge commit performed into master from the react-rewrite branch

Fork/clone (6 projects): the migration in these projects happened in forks or cloned repositories. As usual, the integration of the new code happened through pull requests.

Summary: Most migrations (9 out of 19) happened directly in the master or development branches.

5.2 Survey with Developers

To validate our findings and obtain more insights on the migration process, we conducted a survey with developers of the projects that migrated their front-end framework. The survey questionnaire had five open-ended questions:

1. *Why did you decide to change the front-end framework?*

2. *What were the main barriers faced during the migration?*

3. *We found the migration was performed in X days. [Why exactly was it performed so quickly — Why did it take that long]?*

4. *I found you performed the migration using <strategy>. What were the main challenges and also the benefits of using the <strategy> for this specific purpose?*

5. *Are you satisfied with the new framework?*

Table 9: Factors motivating the migration of front-end frameworks

Factor	**Occurences**
Lack of experts	6
Sub-optimal architecture	4
Lack of support	3
High learning curve	3
Poor maintainability	2
Lack of popularity	1

These questions were sent by email to developers of the 19 repositories who completed the migration. Specifically, we contacted the developers with the highest number of commits in front-end-related files in the migration period. We received 11 responses (response rate of 57.8%), including five responses from systems that migrated from Angular to React, three responses from systems that migrated from Vue to React, and one response in each one of the following migrations: from Angular to Vue, from Backbone to React, and from React to Vue. Next, we discuss these answers. When quoting the answers, we use labels RU1 to RU11 to indicate the respondents.

(Q1) Why did you decide to change the front-end framework?

We identified six key factors that motivate the un-adoption of front-end frameworks in JavaScript. Table 9 summarizes these factors. Since some respondents mentioned more than one factor, the number of occurrences in this table is greater than 11. Lack of experts (6 answers) and the sub-optimal architecture provided by the framework (4 answers) are the most common un-adoption factors cited by them. As examples, we have the following answers:

Often in issues, I saw the message: "I'm not good in Angular so I can't help with PR" or similar. (RU7 - Lack of experts)

Angular 1.x architecture is pretty terrible—it spreads the code across lots of different places with hard to follow $scope variables. (RU8 - Sub-optimal architecture)

(Q2) What were the main barriers faced during the migration?

After analysing the seven answers received for this question, we ended up with three main barriers. First, two developers reported difficulties for re-implementing the user experience with the new framework, as in the following answer.

The main barrier was re-developing the user experience for a new application (RU1)

Another barrier (two answers) relates to the lack of experts in the old framework, as in this answer:

Almost all Angular code was written by people who are not active in the project anymore, so we did not have a lot of in-depth knowledge of what the code does. (RU8)

Conversely, two developers also commented on their lack of experience with the new framework.

No one in the team had a React experience at the time we started the migration, so we had to learn it during the process (RU9)

Finally, one developer reported difficulties using both frameworks during the migration.

I think the main challenge must have been at the beginning, to bootstrap the effort and find a way to make it work with both technologies in parallel so that the move could be done slowly over a year while still delivering new features and value for our users (RU5)

(Q3) The migration was performed in X days. [Why exactly was it performed so quickly] [Why did it take that long]?

First, and interestingly, three developers reported that the migration was performed offline in a local copy and then pushed to GitHub, as in the following answers.

This took many days to complete and was performed off GitHub, in a private repository (RU1)

Although it's a single commit on the repo, the migration took a long time ... It was actually a complete rewrite from scratch in Vue rather than a migration (RU3)

For the developer of a project whose migration took two years, the reason was the need to keep the application evolving.

Mostly because we were unable to freeze other works on the project. So we migrated side-by-side with fixing bugs, implementing new features, accepting PRs, etc (RU9)

For another developer, the migration took a long time (three years) due to the project's monolithic architecture.

It took so long because of the way the application was structured, which was very similar to that of a monolithic application ... which made it very uncomfortable to switch from the old architecture to a simpler one (RU2)

(Q4) I found you performed the migration using <strategy>. What were the main challenges and also the benefits of using the <strategy> for this specific purpose?

In this fourth question, we asked the developers about the challenges and benefits of the migration strategy used by them. Specifically, we explicitly cited in the question the migration strategy used by each project (as described in RQ3). In general, we received answers with very specific motivations. For example, developers migrating using the master/development branch highlighted the challenges of having two frameworks running side-by-side during the migration.

Since the plan was to change the framework slowly and not do a big bang change at once, we had no choice but to make sure the webapp would work correctly with the two frameworks in parallel. For a long time some parts of the app were rendered by Backbone while others were rendered by React ... (RU5)

For all the migration period we used React and AngularJS side by side, migrating the application by small parts. (RU9)

On the other hand, forks/clones are the preferred migration strategy mainly when the project already uses a fork/PR model to accept contributions, as mentioned in the following answers:

All contributors develop in their own forked repositories, and send pull requests to [project]. (RU8)

I did the migration in my private repo and then afterwards merged it into the public repo. As the repo was a fork and I stalled the development for the UI in the main repo there were no big challenges. (R10)

Finally, when using a separate branch, developers sometimes managed to release versions from it, as commented in the following answer:

I migrated on a separate branch and was releasing a bunch of alpha versions from it. When migration was ready, I merged it to master. It was done to simplify the migration burden for end-users. (RU7)

(Q5) Are you now satisfied with the new framework?

Except one, all participants are satisfied with the new framework, as in the following answers:

Absolutely satisfied, I think it gives more freedom and is potentially simpler than Angular since the library-specific syntax is so little (RU2)

Yes, very satisfied with Vue. I find the way Vue organises templates + code to be much more natural and easy to read and maintain than React + JSX (RU3)

Yes! Very satisfied. I'm getting much more contributions + working with React is easier for us. (RU7)

6 Implications

This section presents the implications of our study for practitioners, framework developers, tool builders, and researchers.

For practitioners: our study provides interesting insights for practitioners that are interested in adopting (or un-adopting) a front-end framework in their JavaScript projects. We listed factors that motivate the (un-)adoption of such frameworks and provided quantitative and qualitative data about open-source projects that started and finished a migration effort. Particularly, we showed that such migrations are not trivial efforts, e.g., 5 out of 12 projects spent more time performing the migration than using the old and new frameworks. In the following box, we summarize our (concrete) recommendations for practitioners, both when adopting and when un-adopting a framework.

Recommendations for adoption:

- Key factors: framework's popularity, learnability, and architecture.
- Other factors: previous expertise, community, performance, and support.
- Evaluate the quality of the framework's documentation.
- Analyze whether the framework's architecture fits the application's one.
- Analyze the team's expertise.

Recommendations for unadoption/migration:

- Carefully assess the migration risks, including negative impacts.
- Evaluate a gradual migration approach, when both frameworks are used in parallel during the migration.
- Start the migration by the simplest components.
- Ask from stakeholders feedback, especially on key factors identified in our un-adoption study, such as user experience.
- Involve experts in the old framework in the migration to clarify possible doubts.
- Invest in training the team in the new framework.

For framework developers: our study provides key lessons also to the developers of modern JavaScript frameworks. First, we show that the framework's adoption is guided by some factors they do not directly control. For example, framework developers do not have direct influence on the popularity of their frameworks. On the other hand, we also listed several factors that developers might improve, such as quality of the documentation, performance, and architecture. Finally, our study confirms that front-end frameworks developers should be prepared to compete in a very dynamic ecosystem. For example, 18 (36.7%) out of 49 framework users have plans to use a second framework in their current or future projects.

For tool builders: we also envision a market for developers interested in designing and implementing tools for supporting framework migration and inter-operation. Interestingly, we detected that four projects use a tool called REACT2ANGULAR,[11] which allows importing REACT components in ANGULAR apps. However, it seems such tools are still in their infancy. For example, the mentioned tool does not support component nesting.[12]

For researchers: We also found a limited number of scientific studies on JavaScript front-end frameworks. Since they are fundamental components in modern JavaScript applications, we claim that researchers should also invest more time and effort on aspects such as architecture, design, performance, and documentation of such systems.

[11]https://github.com/coatue-oss/react2angular
[12]https://github.com/coatue-oss/react2angular/issues/11

7 Threats to Validity

We discuss three types of threats to validity [14].

External Validity: The first threat is related to the generalization of our results. In this paper, we presented an analysis of the adoption and un-adoption of JavaScript front-end frameworks. In the un-adoption study, we sent one e-mail for each project out of 19 and received 11 answers (57%). Despite this high response ratio, we agree that the total number is low, and difficult to generalize our results to other projects. Moreover, although we considered five of the most popular and well-known frameworks, we cannot generalize our results to other JavaScript frameworks.

Internal Validity: This threat relates to facets that may affect our experimental results. We relied on information stored in *package.json* and *bower.json* files to identify the projects using the analyzed frameworks. These files include a list of required dependencies. However, we acknowledge it is possible to use the frameworks without a package manager, although this is not a recommended practice when building professional applications.[13][14][15] Additionally, detecting dependencies without package manager support is not trivial and error prone.

Construct Validity: A construct validity threat relates to the selection of candidates for our surveys. In our study, we rely on the maintainers' answers to characterize the adoption and migration of front-end frameworks. However, some of the studied projects have hundreds of contributors with distinct responsibilities. To increase the chances of receiving accurated responses, we adopted a Commit-Based Heuristic over changes in front-end files. Moreover, in the second survey, we considered only commits performed during the migration process. In both cases, we may have missed maintainers who do not contribute with code.

8 Related Work

There is a fair amount of studies on the use and acceptance of technologies. For example, Polančič *et al.* [15] investigated the characteristics and individual differences that impact the users' perceptions about object-oriented frameworks by using the Technology Acceptance Model (TAM) [16]. However, the number of studies that explore the adoption of front-end JavaScript frameworks is limited. In an exploratory study, Graziotin and Abrahamsson [17] claimed that these studies focused on benchmarks and other quantitative metrics, whereas practitioners also have interests in other aspects than those of academic research. For example, they argue that practitioners are driven by different concerns when choosing a JavaScript framework, such as the age of the latest release, the size of the framework, the license, the presence of features, and the browser support. Thus, they propose a comparison framework that combines researchers' interests with practitioners's interests to meet the best of both worlds.

[13]https://vuejs.org/v2/guide/installation.html
[14]https://reactjs.org/docs/create-a-new-react-app.html
[15]https://angular.io/guide/setup-local

Based on these findings, Pano *et al.* [4] conducted a study on factors and actors that explain the adoption of JavaScript frameworks. They relied on semi-structured interviews with 18 developers. By using the Unified Theory of Acceptance and Use of Technology (UTAUT) [18], they ended up with five major groups of adoption factors: (i) performance expectancy (performance, size), (ii) effort expectancy (automatization, learnability, complexity, understandability), (iii) social influence (competitor analysis, collegial advice, community size, community responsiveness), (iv) facilitating conditions (suitability, updates, modularity, isolation, extensibility), and (v) price.

However, their study was conducted in 2014, when front-end frameworks were still in the early adoption phases. For example, the most popular framework nowadays (Vue) was not mentioned by any participant and a single participant mentioned the second most popular framework nowadays (Facebook's React). The characteristics of these new frameworks can influence the factors of adoption. For this reason, we targeted modern and open-sourced front-end frameworks, which are widely used nowadays.

The practitioners in our adoption study also mentioned similar factors addressed by the mentioned study, such as performance, community, and learnability. However, we also found new factors not covered by them, such as architecture, expertise, and sponsorship. Moreover, to the best of our knowledge, we are the first to explore front-end frameworks' migration, providing a quantitative and qualitative characterization of frameworks's migration, which is a relevant and relatively common event in the JavaScript ecosystem.

Other studies evaluate other aspects of the frameworks, which can influence their (un-) adoption. For example, Gizas *et al.* [19] conducted several quality, performance, and validation tests to evaluate general aspects of JavaScript frameworks (i.e., they are not only restricted to front-end implementation). The authors relied on well-established software quality metrics, such as size metrics (e.g., lines of code), complexity metrics (e.g., McCabe's Cyclomatic Complexity), and maintainability metrics (e.g., Halstead metrics). Their quality tests revealed parts of the code that probably need to be improved, while their validation tests focused on parts that must be modified to harmonize with browsers' continuous evolution.

In another study, Mariano [20] compared three well-known frameworks (React, Angular, and Backbone) using benchmarks and complexity metrics. The author implemented a benchmark application (a Todo application) using the three frameworks. Regarding performance, Backbone outperformed the other frameworks in their experiment (probably because it is a lightweight framework with just 6.5 KB plus 43.5 KB of required dependencies). In terms of complexity (measured using a cyclomatic complexity metric), ReactJS has the highest mean per-function measure (1.85), followed by Backbone (1.85) and Angular (1.16).

Ramos *et al.* [21] surveyed 95 professional developers about performance issues of Angular applications and reported common practices to avoid performance problems, including the use of third-party or custom components, inadequate application architectures, or unnecessary processing in the digest cycle, which is the internal computation that automatically updates the view with changes detected in the model. Nakajima *et al.* [22] proposed a playground tool named Jact that enables developers to compare the runtime performance of JavaScript frameworks based on typical tasks in Web development. The authors argue that by sharing tasks and source code written by administrators and developers, Jact can continuously provide information related to JavaScript frameworks, including benchmark and API usage. Zerouali *et al.* [23] empirically analyzed the relationship between different software popularity measures of JavaScript libraries from three open source tracking systems (LIBRARIES.IO, NPM, and GitHub). They report that popularity can be measured using different metrics. Ocariza *et al.* [24] introduced an approach to detect inconsistencies between identifiers in applications implemented using JavaScript frameworks. They analyzed 20 ANGULAR projects and found 15 bugs in 11 of them.

Wei *et al.* [25], performed an empirical study on JavaScript applications to understand the dynamic behavior of objects. The authors defined and evaluated several dynamic metrics on user and native objects. Among their findings, they observed that the use of prototype-based inheritance is not well understood by developers and only a few native objects may affect application behavior. Silva *et al.* [26] analyzed the migration of structures that emulate classes in legacy JavaScript code to the classes syntax introduced by ECMAScript 6. To this purpose, they proposed a set of migration rules and evaluated them in eight legacy JavaScript systems. The authors concluded that tools to migrate to new syntax automatically are challenging and risky. Rudafshani and Ward [27] argued that the migration of application logic to the client side of modern web applications can result in memory leaks and issues. Then, they proposed a tool to support developers in detecting and diagnosing such leaks. They applied the tool on open-source Web applications and were able to find and fix several memory leaks on them.

There are websites that provide data on the popularity of JavaScript libraries and frameworks. For example, the most recent Stack Overflow Survey[16] shows that REACT.JS and ANGULAR are more popular than traditional MVC-based web frameworks, such as Spring, Django, and Ruby on Rails. Another popular site with information on the usage of JavaScript-based technologies is State of JS.[17] The site annually runs a survey including mostly closed questions. On the other hand, the survey we used in Section 4 is based on open questions, which were carefully analyzed and categorized using thematic analysis. Moreover, our sample of participants includes key developers of popular GitHub projects that effectively use the studied frameworks.

Finally, Bajammal *et al.* [28] proposed a technique that employs unsupervised learning of mockups (i.e., an illustration of the anticipated UI design) to create reusable components for the clients of modern JavaScript front-end frameworks. Their results showed an average of 94% precision and 75% recall in terms of agreement with the developers' assessment.

[16]https://insights.stackoverflow.com/survey/2020#technology-web-frameworks
[17]https://stateofjs.com

9 Conclusion and Future Work

JavaScript is the language that runs the Web. In the large, complex, and dynamic ecosystem created around the language, a remarkable class of applications are the frameworks widely used by front-end JavaScript developers to architecture and implement rich Web apps, usually called *Single-Page Applications*. In this paper, we studied the factors that drive the adoption of five of such frameworks. We found that the two main factors are popularity and learnability. Furthermore, we also studied the migration of JavaScript frameworks. We showed that lack of experts and sub-optimal architectures are the main factors influencing maintainers to migrate to another framework. Finally, in Section 6, we derived a set of implications of our study for practitioners, framework developers, tool builders, and researchers.

As future work, we plan to study front-end frameworks' internal architecture, aiming to document the key design and architectural patterns used by them. We envision that this study can help practitioners gain an in-depth understanding of the internals of such frameworks and, therefore, better assess the trade-offs involved in their adoption. We also envision space for a survey with specific questions on framework adoption that could support a direct comparison among the most popular frameworks. For example, this new survey should include questions asking the developers about the advantages they find in the API, documentation, or community of a given framework X when compared to another framework Y.

Additionally, we plan to investigate and improve the state-of-the-practice on tools for supporting developers on front-end frameworks migration.

Our datasets—including the survey responses in an anonymized format—are available at: https://doi.org/10.5281/zenodo.4148591

Acknowledgements

Our research is supported by FAPEMIG and CNPq.

References

[1] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *13th International Conference on Mining Software Repositories (MSR)*, 2016, pp. 351–361.

[2] M. Mikowski and J. Powell, *Single page web applications: JavaScript end-to-end.* Manning Publications Co., 2013.

[3] M. Ramos, M. T. Valente, R. Terra, and G. Santos, "AngularJS in the wild: A survey with 460 developers," in *7th Workshop on Evaluation and Usability of Programming Languages and Tools*, 2016, pp. 9–16.

[4] A. Pano, D. Graziotin, and P. Abrahamsson, "Factors and actors leading to the adoption of a JavaScript framework," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3503–3534, 2018.

[5] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view-controller user interface paradigm in smalltalk-80, ch. 31 (3)," *Journal of Object-Oriented Programming*, vol. 1, no. 3, pp. 26–49, 1988.

[6] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of GitHub repositories," in *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 334–344.

[7] H. Silva and M. T. Valente, "What's in a GitHub star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, no. 1, pp. 112–129, 2018.

[8] S. Baltes and S. Diehl, "Worse than spam: Issues in sampling software developers," in *10th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2016, pp. 1–6.

[9] M. Joblin, S. Apel, C. Hunsen, and W. Mauerer, "Classifying developers into core and peripheral: An empirical study on count and network metrics," in *39th International Conference on Software Engineering (ICSE)*, 2017, pp. 164–174.

[10] G. Robles, J. M. Gonzalez-Barahona, and I. Herraiz, "Evolution of the core team of developers in libre software projects," in *6th International Working Conference on Mining Software Repositories (MSR)*, 2009, pp. 167–170.

[11] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2011, pp. 275–284.

[12] D. S. Cruzes and T. Dybå, "Research synthesis in software engineering: A tertiary study," *Information and Software Technology*, vol. 53, no. 5, pp. 440–455, 2011.

[13] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[14] C. Wohlin, *Experimentation in software engineering.* Springer Science & Business Media, 2012.

[15] G. Polančič, M. Heričko, and L. Pavlič, "Developers' perceptions of object-oriented frameworks – An investigation into the impact of technological and individual characteristics," *Computers in Human Behavior*, vol. 27, no. 2, pp. 730–740, 2011.

[16] F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology," *MIS Quarterly*, vol. 13, no. 3, pp. 319–340, 1989.

[17] D. Graziotin and P. Abrahamsson, "Making Sense Out of a Jungle of JavaScript Frameworks," in *Product-Focused Software Process Improvement*, 2013, pp. 334–337.

[18] V. Venkatesh, M. G. Morris, G. B. Davis, and F. D. Davis, "User acceptance of information technology: Toward a unified view," *MIS Quarterly*, vol. 27, no. 3, pp. 425–478, 2003.

[19] A. Gizas, S. Christodoulou, and T. Papatheodorou, “Comparative evaluation of javascript frameworks,” in *21st International Conference on World Wide Web*, 2012, pp. 513–514.

[20] C. Mariano, “Benchmarking JavaScript Frameworks,” Master’s thesis, Technological University Dublin, 2017.

[21] M. Ramos, M. T. Valente, and R. Terra, “AngularJS performance: A survey study,” *IEEE Software*, vol. 35, no. 2, pp. 72–79, 2018.

[22] N. Nakajima, S. Matsumoto, and S. Kusumoto, “Jact: A playground tool for comparison of JavaScript frameworks,” in *26th Asia-Pacific Software Engineering Conference (APSEC)*, 2019, pp. 474–481.

[23] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, “On the diversity of software package popularity metrics: An empirical study of npm,” in *26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 589–593.

[24] F. S. Ocariza, K. Pattabiraman, and A. Mesbah, “Detecting Inconsistencies in JavaScript MVC Applications,” in *37th IEEE International Conference on Software Engineering (ICSE)*, 2015, pp. 325–335.

[25] S. Wei, F. Xhakaj, and B. G. Ryder, “Empirical study of the dynamic behavior of JavaScript objects,” *Software: Practice and Experience*, vol. 46, no. 7, pp. 867–889, 2016.

[26] L. H. Silva, M. T. Valente, and A. Bergel, “Refactoring legacy JavaScript code to use classes: The good, the bad and the ugly,” in *16th International Conference on Software Reuse (ICSR)*, 2017, pp. 155–171.

[27] M. Rudafshani and P. A. S. Ward, “Leakspot: detection and diagnosis of memory leaks in JavaScript applications,” *Software: Practice and Experience*, vol. 47, no. 1, pp. 97–123, 2017.

[28] M. Bajammal, D. Mazinanian, and A. Mesbah, “Generating reusable web components from mockups,” in *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 601–611.