# Why and How JavaScript Developers Use Linters

*Master's Thesis*

Kristín Fjóla Tómasdóttir

# Why and How JavaScript Developers Use Linters

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Kristín Fjóla Tómasdóttir
born in Reykjavík, Iceland

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

# Why and How JavaScript Developers Use Linters

Author: Kristín Fjóla Tómasdóttir
Student id: 4518063
Email: `k.f.tomasdottir@student.tudelft.nl`

**Abstract**

A linter is a type of static analysis tool that warns software developers about possible errors in code or violations to coding standards. By using such a tool, errors can be surfaced early in the development process when they are cheaper to fix, and code can be kept more readable and maintainable. For such a tool to be successful, it is important for its creators to understand the needs and challenges of developers when using a linter. Furthermore, it needs to be made clear to developers why using such a tool can be beneficial, along with how linters can be configured to identify appropriate and relevant issues for their projects.

In this thesis, we examine developers' perceptions of linters to increase our knowledge on these tools for JavaScript, the most widely used programming language in the world today. More specifically, we study why and how developers use ESLint, the most popular JavaScript linter, along with the challenges that they face while using the tool. We collect data with three different methods where we first interview 15 experts on using linters, then analyze over 9,500 ESLint configuration files and finally survey more than 300 developers from the JavaScript community. The combined results from these analyses provide developers, tool makers and researchers with valuable knowledge and advice on using and developing a linter for JavaScript.

Thesis Committee:

| | |
|---|---|
| Chair: | Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft |
| University supervisor: | Dr. M. F. Aniche, Faculty EEMCS, TU Delft |
| Committee Member: | Prof. Dr. G. Gousios, Faculty EEMCS, TU Delft |
| | Prof. Dr. C. Hauff, Faculty EEMCS, TU Delft |

# Preface

This thesis is the product of my graduation project to obtain a master's degree in Computer Science at Delft University of Technology. I would like to thank my supervisor Arie van Deursen for his invaluable advice over the past year. Many thanks go to the participants of this study, those who took part in the interviews and those who spent time to pilot test my survey, and in particular Joseph Hejderup for his valuable input. Special thanks go to my second supervisor, Maurício Aniche, for his endless help and enjoyable collaboration during this project. Not only did he always make time to answer all my questions and review my work, but also gave me moral support and continued to encourage me throughout my project.

Lastly, I am lucky to have been able to spend the two years of my MSc studies in Delft, where I have gotten to meet many amazing people from all over the world. From both friends and colleagues I have learned countless lessons that have influenced my greatly as a person, for which I am ever thankful.

Kristín Fjóla Tómasdóttir
Delft, the Netherlands
August 28, 2017

# Contents

# List of Figures

# Chapter 1

# Introduction

An important part of software development is to maintain code by keeping it readable and defect free. A well known method to do so is using automatic static analysis tools (ASATs) which automatically examine code to look for defects or any issues related to best practices or code style. These tools aid in finding issues and refactoring opportunities early in the software development process, when they require less effort and are cheaper to fix [73, 105]. Due to their many benefits, ASATs have become commonly used in software development [69].

There is an abundance of available ASATs, both academic tools and tools used in industry. These tools vary in functionality, use diverse approaches for static analysis and can work for different languages [139]. Some tools focus on coding styles, code smells or general maintainability issues, while some others try to identify faults in code, sometimes examining specific types of defects such as related to security or concurrency [169, 74]. One type of ASATs are linters, which often use a relatively simple analysis method to catch non-complex errors and violations to coding standards.

In recent years, linters have become commonly used tools for dynamic languages such as JavaScript [69]. JavaScript has become a very popular programming language in the last years and in fact has been the most commonly used language on GitHub since 2013 [91]. It is known as the language of the web and has recently also become popular for server side development, serving as a general-purpose language. A notable characteristic of JavaScript is its dynamic nature. For example, it allows for generating new code during runtime execution and for dynamic typing where variables do not need to be declared before they are used. Partly due to its dynamic features, JavaScript is considered to be an error-prone language [129]. For example, it can be easy to introduce unexpected program behavior with simple syntactic or spelling mistakes, which can go unnoticed for a long time [128, 133]. A linter can therefore be especially useful for JavaScript to detect these types of mistakes. Additionally, as JavaScript has become an extremely widespread language, it becomes even more important to have tool support that aids developers in keeping JavaScript code maintainable, secure and correct.

Several studies have focused on static analysis for JavaScript since ASATs have different kinds of requirements for dynamic languages than for static languages [126, 111, 86]. Other research has been conducted on general ASATs, including how developers use these tools and perceive them [64, 108, 74, 69]. Using ASATs does not come without its challenges. For example, they are known to produce a high number of warnings which includes many false positives [108, 71]. Some of these warnings are often not relevant for all projects and can therefore be perceived as false positives when tools are not configured appropriately [67, 105, 74]. In the past, researchers and industrial developers have had different assumptions and requirements when it comes to the usage of ASATs [71]. As has been attempted in previous research [74, 108], it is important to study developers' perceptions of these tools to narrow the gap between academic research and industrial use, in order to make future research more relevant and meaningful. Furthermore, with additional knowledge of how developers use and perceive these tools, ASAT creators can acquire the appropriate context and requirements for future development and improvement. For a widely used language such as JavaScript, this becomes especially important for a tool to gain popularity in such a large community.

Until now, no studies have concentrated on how developers perceive ASATs specifically for a dynamic language such as JavaScript. As it is very different from other commonly studied programming languages, such as Java, we expect different motivation and behavior from developers. In this study, we aimed at understanding why and how developers use linters in real world JavaScript applications. We specifically investigate the features and functionalities that developers find important and the challenges that they face when using linters. Furthermore, linters need to be incorporated to the development process and configured appropriately for a project. This can be done in different ways and can be a demanding process when there are many rules to choose from. We investigate what methods developers use to configure linters and how they maintain those configurations. For this purpose we examine the usage of ESLint [11], the most commonly used linter for JavaScript[1]. As ESLint offers the greatest amount of functionality and flexibility out of all well known linters, it was chosen for this study to avoid excluding or focusing on any specific type of linting such as only analyzing styling issues or possible defects.

For this study we use a mixed methods research approach which involves collecting a combination of qualitative and quantitative data [78]. First we apply a qualitative method, inspired by Grounded Theory [93], to conduct and analyze interviews with 15 developers from reputable open source software (OSS) projects. These developers were identified to be actively involved with enabling and configuring ESLint in the corresponding projects. Next we perform a quantitative analysis on the usage and configurations of ESLint in 9,548 JavaScript projects on GitHub. Finally, to challenge and generalize the previous two analyses, we survey 337 developers from the JavaScript community. We ask them about their experiences and perceptions with using linters, employing the previously acquired knowledge as input to the questionnaire.

The main contributions of this thesis are the following.

- A qualitative study on the perceptions of ESLint by interviewing 15 developers that have actively used and configured the tool in reputable OSS projects (Chapter 4). This chapter was conditionally accepted as a paper for the proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017).

[1]According to the number of npm downloads [162].

- An extensive quantitative analysis of linter configurations in over 9,500 JavaScript projects (Chapter 5).
- A study on the experiences and perceptions of linters, and the importance of various ESLint warnings, by surveying over 300 JavaScript developers (Chapter 6).

The structure of the thesis is as follows: We start by reviewing background information in Chapter 2 and related work in Chapter 3, in order to provide an appropriate context for the remaining text. Following is the main body of the thesis which consists of three chapters. Each chapter contains its research questions, methodology and results, and finally a discussion on the analysis along with a short conclusion. We finally conclude the thesis in Chapter 7 where we revisit all research questions, discuss the implications for the complete study and end with a final conclusion.

# Chapter 2

# Background

In this chapter we provide information on the languages and tools that concern this study, namely JavaScript, ESLint and other available JavaScript linters.

## 2.1 JavaScript

In the following we examine the history of JavaScript, the nature and environment of the language along with its key features.

### 2.1.1 History of JavaScript

JavaScript was created in 1995 by Brendan Eich with a goal to add scripts into static HTML documents for web pages [95]. The language was developed for the first popular browser, Netscape, originally under the names Mocha and LiveScript [144]. At that time, the future of web pages was visioned to be in the hands of Java applets, so in order to make this new scripting language more appealing to the public, it was partly named after Java and made to have similar syntax [95, 144]. Intentionally, the language itself was constructed to be not too similar to Java, as JavaScript was supposed to be a lightweight complimentary "*silly little brother language*" to Java [144, 121]. The reference to Java in the name JavaScript has created many misconceptions that these two languages are related, where in fact they are very different [95].

The official name of the language is actually ECMAScript, where JavaScript is technically only a specific implementation from Netscape and the Mozilla foundation [88]. Other implementations of JavaScript include JScript from Microsoft but these different names were derived as it was not possible to get a license from Sun (now Oracle) for the language, who had a trademark on the term JavaScript [136, 121]. The language is in general however commonly known as JavaScript (and will be used in the remaining text), whereas the name ECMAScript is more commonly used when referring to the official specifications and versions of JavaScript. These different names have indeed caused some confusion and discontent, where Eich described the name ECMAScript as something that "*sounds a little like a skin disease*" and that "*nobody really wants it*" [121]. Since 1996, the language has been standardized by the organization Ecma International [136]. The specification is called

ECMA-262 and is managed by a technical committee of Ecma, consisting of members from notable companies such as Microsoft, Mozilla and Google. The first ECMAScript version, ECMAScript 1 [101], was released in June 1997, later followed by additional seven versions. The latest versions are the sixth [103] and seventh [104], released in 2015 and 2016, respectively, going by the names ES6 (or ES2015) and ES7 (or 2016).

At the time when JavaScript was first created, the pace of innovation for the web was extremely fast. Eich was therefore required to develop a prototype for the language in a very short time, and succeeded to do so in only 10 days [144]. The goal of the language of that time was to be a simple lightweight scripting language and was originally intended for less experienced developers, such as designers, to use to add small features to web pages [136, 121]. The early edition of JavaScript was immediately included in Netscape Navigator 2 in a very rough version with some debatable design choices, but it quickly became popular as the language of the web [79, 144]. In the following years, more browsers became popular with their development occurring faster than any web standards could follow [95]. Even with standards being up-to-date, different browsers implement them in different ways. It therefore became difficult for developers to create a consistent experience for web page users across different browsers and operating systems due to these compatibility issues. Most notably, browsers implemented the Document Object Model (DOM) in different ways, *e.g.*, with Internet Explorer often lacking many elements defined in W3C DOM standards. While the different JavaScript implementations also caused some problems, it was much less problematic than the differences in DOM implementations [136, 148] [1]. All these previously mentioned facts played their part in creating a bad reputation for JavaScript, making it one of the most controversial programming languages. However, since its first edition, the language has grown and improved through its many versions and also with the help of external libraries [121, 144]. Today it is one of the most popular programming languages that exist. It has been the most commonly used language on GitHub since 2013 [91, 45] and for five years in a row the most commonly used language by developers participating in the annual Stack Overflow survey [130].

### 2.1.2 JavaScript's Ecosystem

JavaScript has most commonly been used for the browser as a client-side language [88]. While the original goal of the web was to read and write information, it has become much more functional and interactive with the presence of JavaScript [70, 136]. Scripts are embedded to HTML pages where they manipulate the document content through the DOM [88]. Cascading Style Sheets (CSS) are then commonly used to modify the presentation of the web page's content. Ajax technologies, short for asynchronous JavaScript and XML, are used to load new content dynamically without reloading a web page. It uses scripted HTTP to exchange data with a web server to facilitate interactive behavior. Furthermore, there are many libraries that make client-side JavaScript development an easier activity, such as the popular jQuery library [35] to help with DOM manipulation. More recent popular libraries

[1] Today, the lack of browser support for new JavaScript features can be remedied with JavaScript compilers such as Babel [2] which translates newer JavaScript into older JavaScript that the browser supports.

and frameworks include AngularJS [1], Ember.js [10] and React [49], intended to simplify the creation of web applications.

As JavaScript is a general-purpose language, it can also be used outside the browser, *e.g.*, for server side development. This has become more common with the introduction of the run-time environment Node.js [43], which facilitates writing server-side code and shell scripts [149]. JavaScript can be used in other environments as well, *e.g.*, to create mobile applications with Facebook's recent framework React Native [51] or desktop applications with Electron [9]. As the JavaScript community grows, more tools and libraries start emerging, such as build tools and testing frameworks. The package manager npm [44] then emerged with Node.js which allows developers to easily distribute and install JavaScript modules.

Finally, JavaScript started out as an interpreted language with the first JavaScript engine, SpiderMonkey [52]. JavaScript was previously known to be a relatively slow language but there have been major improvements in speed in recent years [136]. Faster just-in-time JavaScript engines have emerged, such as Firefox's TraceMonkey [56] and Google's V8 [60], replacing older and slower JavaScript interpreters.

### 2.1.3 JavaScript's Nature and Language Features

As touched upon before, although JavaScript is partly named after Java, the two programming languages are not so similar. However, certain elements in JavaScript were indeed inspired by Java such as the language syntax and the separation of values into primitives and objects [136]. JavaScript was influenced by several other languages as well in its creation. Some functional properties of the language were borrowed from Scheme, including the presence of first-class functions and closures. The implementation of object orientation with prototype-based inheritance was inspired by Self while JavaScript's functions were influenced by AWK. Finally, Perl and Python influenced the handling of strings, arrays and regular expressions. With all these different influences and features, JavaScript becomes a versatile and a powerful language [79].

Perhaps the most distinctive and notorious aspect of the JavaScript language is its dynamic nature. The language allows for generation of new code during runtime execution, such as with the infamous `eval` function. This function takes a string as an argument which can represent any kind of expression or statement (or multiple) to be executed, and is used frequently in JavaScript programs [138]. Another dynamic feature is the fact that many things can be changed [136]. Objects' properties can be dynamically added and removed at any time, opposed to only being set at object creation. Furthermore, these properties along with variables in general, can have different types during a program's lifetime. This is commonly referred to as dynamic typing where the types of variables are unknown at compile time, opposed to static typing. This aspect of JavaScript is considered to be flexible and convenient by many developers, where it is possible to write concise code in shorter time without having to worry about any type information [148, 79, 133]. Dynamic typing is however known to relate to many subtle errors that can occur around inconsistent use of types [133]. Moreover, JavaScript performs silent type coercion which can result in unexpected behavior that can be difficult to identify [148, 133]. Another example is when a variable name is mistyped where no error is thrown but a new global variable is created instead

[128]. Several prominent companies have spent effort on these issues and have published static type systems for JavaScript which include Google's Closure [6] and Facebook's Flow [30]. Perhaps the most notable type checker for JavaScript is Microsoft's TypeScript [59], a superset of JavaScript which allows for optional static typing.

Since version ES6 from 2009, it is possible to use JavaScript in `strict mode` which is a restricted and slightly different version of the language [102, 136]. When enabled, errors are thrown for various items that normally only cause silent errors. In strict mode, all variables need to be declared, all function parameter names need to be unique and the functionality of `eval` is restricted. These and other limitations are intended to prevent common errors in JavaScript.

## 2.2 JavaScript Linters

Static analysis tools are used to automatically examine code and look for defects or any issues related to best practices or code style, without executing any code. One of the first widely used ASATs was a linting program for C in 1977 [109]. More recent and well known ASATs include FindBugs [29], Checkstyle [5] and PMD [46]. While FindBugs is focused on discovering code defects and Checkstyle targets coding styles, PMD detects both defects and violations of coding standards. Static analysis tools for JavaScript are commonly referred to as linters where they also have similar roles as described above. As JavaScript programs can be large and complex [135, 138], and where JavaScript code can be error-prone and difficult to keep maintainable [128, 129, 86], linters can be a very useful addition to the development process. In the following we discuss the available linters for JavaScript and more thoroughly examine ESLint, the most popular JavaScript linting tool.

### 2.2.1 Available Linters

JSLint [38] was the first available linter for JavaScript, created by Douglas Crockford in 2002 [79]. The tool analyzes source code to detect potential errors and style violations according to Crockford's opinions on how JavaScript should be written, and with that eliminating the "bad parts" of the language. JSLint is an open source software on GitHub [40] but is solely maintained by Crockford. As JSLint was not considered to be configurable enough (and to be overly opinionated) [120], JSHint [37] was created in 2010 as a fork of JSLint to be a community driven and a more flexible linter for JavaScript. JSHint also includes a richer set of rules than JSLint but is now migrating to focus mostly on code correctness rather than code style. Other linters have followed such as JSCS [36] and ESLint [11], both created in 2013. JSCS is mainly focused on code style while ESLint has a larger variety of rules, covering both code styles and possible defects. However since April 2016, the maintainers of JSCS have joined forced with ESLint and thus not supporting JSCS anymore. Another recent linter from 2015 is Standard JS [53] that, unlike the previous linters, is not configurable. It is built upon ESLint but has a fixed configurations for the rules that ESLint provides.

All these linters are open source and available via npm. Figure 2.1 shows the popularity of these linters according to the number of npm downloads since the start of 2015. ESLint is by far the most popular linter where the total number of npm downloads is over 72M, and JSHint the second most popular with approximately 56.5M downloads. Of all these linters, ESLint also has the most active community around it where it has the most contributors on GitHub [25], highest number of commits and frequent releases. As ESLint is a commonly used tool within the JavaScript community and offers various functionality, it was chosen to be the tool of this study. That way we do not focus on any specific type of analysis and obtain the most generalizable results.

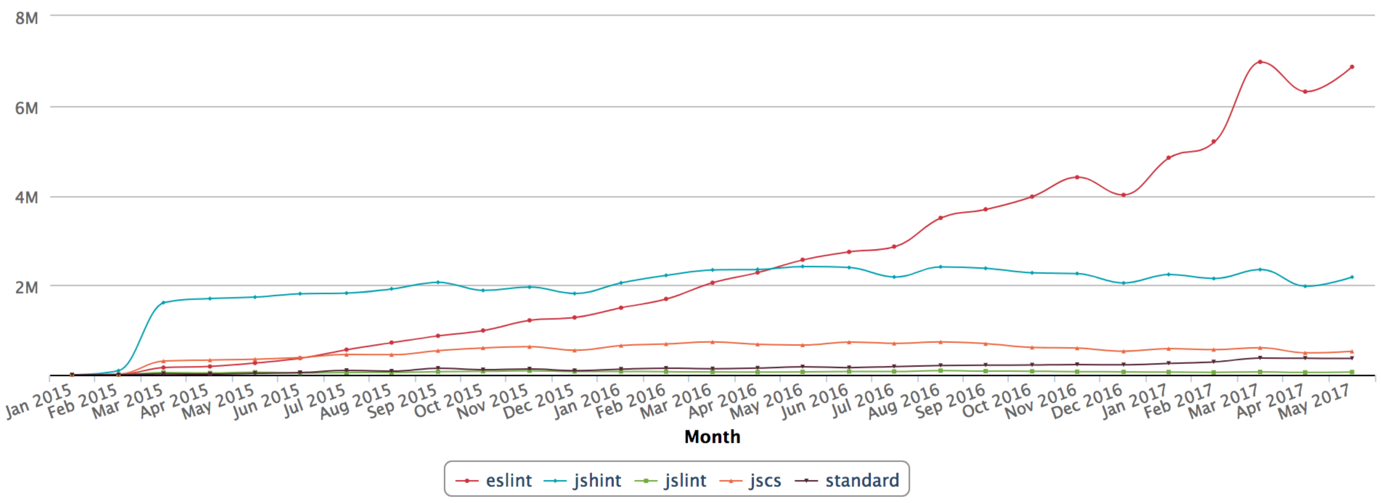


Figure 2.1: Number of npm downloads for ESLint, JSHint, JSLint, JSCS and Standard from January 2015 - May 2017 [162]

Another recent tool from early 2017 is gaining popularity which is called Prettier [48]. This tool is not a linter but an "*opinionated code formatter*", where it solely enforces style based rules and automatically formats code to adhere to them. Their goal is for development teams to completely stop spending time on discussing code style and to have as much automated as possible. Similar to Standard but less popular is the linter XO [61]. It is also built upon ESLint with default configurations, but with an option of being customizable, with an intention of being very easy and fast to use. Furthermore, for TypeScript there is TSLint [57], a configurable linter that analyses code for style and possible errors. Finally, EditorConfig [8], although not a linter, can be used for various programming languages, such as JavaScript, and for different editors to maintain a consistent code style.

### 2.2.2 ESLint

ESLint was created by Nicholas C. Zakas in June 2013 [82]. It is an open source software system available via GitHub [25] with a vibrant community of over 500 contributors and over 5,000 commits. The tool is written with Node.js and is available via npm [26], with 4-7 million monthly downloads in 2017 [162]. It was originally designed to be an extremely flexible linter, where it is both easily customizable and pluggable. More specifically, one can configure the base rules in multiple ways or create custom rules in the same manner as the base rules are written. The tool is used for both error and style checking where some warnings can be automatically fixed by the tool. ESLint uses Espree [27] to parse code into an Abstract Syntax Tree (AST) which is then traversed with Estraverse [28] to execute the rules [83]. By first creating an AST, ESLint is slower in execution than JSHint which evaluates the code while it is being parsed [25]. More details on the implementation of ESLint are not relevant for this study, but rather how the tool can be used by developers. The following sections describe the basic rule set that ESLint provides along with how it is possible to configure the tool.

#### 2.2.2.1 Types of Rules

ESLint comes with a set of 236 base rules[2], grouped in the following categories: *Possible Errors, Best Practices, Strict Mode, Variables, Node.js & CommonJS, Stylistic Issues and ECMAScript 6* [85] (categories are listed in the order as they are mentioned in the ESLint documentation). Table 2.1 further describes each of these categories. In order to get a better feeling of what these categories and rules are about, we describe some example rules for each of the seven categories. There are two example rules per category in the following, with a description according to their documentation [85] and other resources.

**Possible Errors**

- `no-console`: Disallows usage of logging to the console, such as the statement `console.log("text");`. Logging statements are mostly used for debugging purposes which should not be present in production code. These statements are also known to causer errors in some old browsers and should thus be removed.

- `no-cond-assign`: Disallows assignment operators in conditional statements such as `if`, `for` and `while` statements. This is done as it can be easy to unintentionally type an assignment operator (=) instead of a comparison operator (==), in which case most likely results in unexpected behavior of an application.

**Best Practices**

- `no-eval`: Disallow usage of `eval()`. The notorious "evil eval" statement is one of the methods that JavaScript provides to execute code at runtime and is known to be one of the most misused features of the language [79]. Using eval is considered bad practice as it can slow down the performance of an application, compromise its security and does not welcome any static analysis [137].

- `eqeqeq`: Require use of type safe equality operator (===) over the regular equality operator (==). The regular equality operator uses type coercion in the comparison which can produce some unexpected results when comparing two items with different types, and is therefore considered as a bad practice [88, 79].

[2]As of release v3.13.0 in January 2017. Rules added in more recent releases are not considered in this study.

### Strict Mode

- `strict`: Require or disallow strict mode directives, and is the only rule in this category. Writing the statement `"use strict";` at the start of a file or a script activates the strict mode of JavaScript which disallows the use of certain features and throws errors for certain elements that otherwise result in silent errors, such as using undeclared variables [102].

### Variables

- `no-undef`: Disallow undeclared variables. This rules prevents possible errors that can result from mistyping a variable name in which case a new global variable is created instead.

- `no-unused-vars`: Disallows unused variables. Unused variables can take up unnecessary space in the code if left behind and can lead to confusion, and should thus be eliminated. Moreover, they can also be a source of errors in the case where a variable name is misspelled.

### Node.js & CommonJS

- `no-path-concat`: Disallow string concatenation using global Node.js variables. In Node.js, the path to the directory and file paths of the currently executing script are kept in the global variables `__dirname` and `__filename`. These are commonly used with string concatenation to reference other directories. This is a bad method to do so as different operating systems interpret the paths differently and can thus create invalid paths. Node.js offers a specific module to perform this task instead of using string concatenation.

- `global-require`: Enforce dependency declarations on the top-level module scope. Module dependencies in Node.js are included with a `require()` function which can be placed anywhere in the code. Placing these calls at the top-level of a module makes it easier to identify all dependencies, and can also prevent possible synchronization problems when including modules.

### Stylistic Issues

- `indent`: Enforce consistent indentation. Indentation has long been a debate between developers, whether one should use tabs or spaces, and even then how many spaces [167]. Having inconsistent indentation can make code more difficult to read and should thus be enforced.

- `comma-dangle`: Require or disallow trailing commas in object literals. A trailing comma can be placed after the last property-value pair in an object literal, even though it is not followed by any additional properties. This can cause an error in some versions of Internet Explorer and should thus maybe be disallowed. However, having a trailing comma can make code changes simpler and easier to review, as fewer lines need to be modified when changes are done to an object.

**ECMAScript 6**

- `no-const-assign`: Disallow modification of `const` variables. Along with ES6 came the `let` and `const` statements to declare a variable, additionally to the previous `var` statement. The `const` statement is intended for single-assignment variables and, while it is not immutable, an error will be thrown if the variable is reassigned. However, in non ES6 environment this can be ignored and no error will be raised.
- `no-dupe-class-members`: Disallow duplicate names in class members. ES6 introduced a new syntax to define objects which is more similar to classes in Java. When two or more methods in an object have the same name, the last method declaration will overwrite the others, possibly causing unexpected behavior in an application.

| **Category** | **Description** | **Available rules** |
|---|---|---:|
| Possible Errors | Possible syntax or logic errors in JavaScript code | 31 |
| Best Practices | Better ways of doing things to avoid various problems | 69 |
| Strict Mode | Strict mode directives | 1 |
| Variables | Rules that relate to variable declarations | 12 |
| Node.js and CommonJS | For code running in Node.js, or in browsers with CommonJS | 10 |
| Stylistic Issues | Stylistic guidelines where rules can be subjective | 81 |
| ECMAScript 6 | Rules for new features of ES6 | 32 |
| **Total** | | **236** |

Table 2.1: ESLint rule categories with ordering and descriptions from the ESLint documentation [85]

#### 2.2.2.2 Configurations

Unlike most other general ASATs, ESLint does not come with any default configurations. It is instead extremely customizable, where users can not only configure each rule they want to include, but also which environment the code is run in and which parser is used [84]. By choosing an environment, such as `node` or `meteor`, predefined global variables (specific to each environment) are defined. It is also possible for different language options to be supported for the parser, such as to support JSX or different versions of ECMAScript. It is even possible to define another parser than the default Espree, such as the Babel parser [3] to include experimental JavaScript features that are not supported by ESLint.

These configurations, and others, are specified within an ESLint configuration file. This file should be named `.eslintrc` with one of the following file endings: `.js, .yaml, .yml` and `.json` (with this order of priority when the tool automatically looks for the file). In previous versions, the configuration file did not have any file ending which is still allowed today but is a deprecated format. Alternatively, it is possible to define the configurations in the `package.json` file within an `eslintConfig` property. If the configuration file is placed in the root directory of the project it is applied to all project files. It is also possible to place

```
{
  "env": {
    "es6": true,
    "node": true
  },
  "ecmaFeatures": {
    "jsx": true
  },
  "extends": "eslint:recommended",
  "plugins": [
    "react"
  ],
  "rules": {
    "no-console": 1,
    "indent": [
      2,
      "tab"
    ]
  }
}
```

Listing 2.1: Example configuration file for ESLint

the file in a subdirectory, which is then used in combination with a configuration file in a parent directory, but with the subdirectory having a priority over the other. With that feature it is possible to have different configurations for different parts of a project. Additionally, one can define a file with the name `.eslintignore` which specifies directories or types of files for the linter to ignore. If these naming conventions are used, the linter automatically finds these files and applies them when running the linter. It is however also possible to have custom file names by passing the paths of the files to the linter with command line options when running the tool. An example configuration file is shown in Listing 2.1, containing various types of configurations that are mentioned in this chapter.

There are several different ways to configure rules for ESLint. The main methods are the following:

1. Specify individual rules in configuration file
2. Specify individual rules with comments in a file containing code
3. Use a preset
4. Use a plugin

When specifying an individual rule, it is possible to disable a rule (with the settings `off` or `0`) or to enable it, either as a warning (`warn` or `1`) or an error (`error` or `2`). The

```
/* eslint indent: ["error", 2], curly: "warn" */
```

(a) Enabling specific rules

```
/* eslint-disable indent, curly */
```

(b) Disabling specific rules

```
/* eslint-disable */
```

(c) Disabling all rules

Listing 2.2: Enabling and disabling rules with comments

rule is turned off when applying `off` and will have no effect whatsoever. When a rule is set to `warn`, each instance of the rule being broken will appear in the output when running the linter. Lastly, `error` will have the same effect as `warn` except it gives an exit code of 1, meaning it can break the build when ESLint is a part of the build process. Additionally, some rules have more detailed options that further specify how the rule is applied. As an example, as illustrated in Listing 2.1, the rule `indent` can have the setting `tab` to enforce the usage of tabs for indentation, or the setting `2` to enforce using two spaces. Even further customization is possible with this rule by enforcing explicit indentation settings for different statements and expressions, such as switch statements or variable declarations. The severity described before should always be the first option in a rule's setting which can then be followed with these additional settings.

Just as the rules can be configured in a configuration file that can cover all subdirectories, they can be configured locally to apply only for one file or a specific piece of code. Rules can be enabled with a comment as demonstrated in Listing 2.2. In the same manner, it is possible to disable a specific rule for a file or a piece of code, or even disable all rules as further shown in Listing 2.2

Another way to specify the rules that are enabled is by using a preset. A preset is a sharable configuration package that is made publicly available via npm, with a name in the style of `eslint-config-<name>`. Which particular preset is used, or even multiple presets, can be specified with the `extends` property in the configuration file. A well known preset is `eslint-config-airbnb` [13] which has, together with `eslint-config-airbnb-base` [12], almost 25 million npm downloads since their release in July 2015 [161]. Other linters that were previously discussed, Standard, XO and Prettier, also offer a preset with their default rules to be used with ESLint. With this `extends` option, it is also possible to enable the set of recommended rules from ESLint, `eslint:recommended`. When using a preset, one can override any rule setting, *e.g.*, by disabling some rules that are included in the preset or by enabling some rules that are not included.

Yet another way of including rules in a project is to specify a plugin. While presets are configurations for rules that ESLint provides, a plugin is a set of custom rules that are made publicly available via npm, with the name format `eslint-plugin-<plugin-name>`. One or more plugins can be specified within a `plugins` attribute in a configuration file, or with command line options. To then include all the rules in the plugin, it should be specified along with other presets in the `extends` property, written as such: `plugin:<plugin-name>`. If one wants to use only specific rules from the plugin, they should be specified along with other rules in the `rules` attribute, referenced with: `<plugin-name>/<rule-name>`.

Lastly, it is possible to run an initialization wizard when setting up ESLint, where the tool analyzes the project's source code and asks a few questions. Based on the results, a configuration file is created with a suitable configuration.

# Chapter 3

# Related Work

Static analysis has been a popular research topic where many tools have been created, both for academic and industrial use. These tools can be convenient as they are used without executing the software which they are applied on. A common research topic is the construction of such tools, such as for security vulnerability or fault detection [125, 75] and the various methods to do so, such as data flow, information flow, path or pointer analysis [169]. Other research topics explore the usage of these tools, which we are more interested in for this particular study. These topics include developers' perceptions of ASATs [74, 108], the value of using these tools in real world applications [164, 169] and the challenge of false positives [71, 100]. In the following we briefly explore these latter topics and finally discuss static analysis in the world of JavaScript.

## 3.1 Perceptions of Static Analysis Tools

Johnson *et al.* [108] researched the usage of ASATs, in particular why some developers do not use these tools to find bugs despite of their proven benefits. They conducted a study where they interactively interviewed 20 participants while applying FindBugs on software. They found that the main reasons why the participants chose to use static analysis tools was to avoid the time consuming manual labor to find bugs, to support team development efforts and to enforce coding standards. Reasons to not use these tools include poorly presented output where there are too many false positives or too many warnings reported in general, in addition to tools not being integrated conveniently in the workflow. Moreover, most participants claimed that it is important for these tools to be easily customizable and that warnings should be explained properly including how they can be fixed. Our work is inspired by this study where we also research why static analysis tools are used and how they can be improved, but focusing solely on JavaScript.

A similar study was conducted by Christakis and Bird [74] who also investigate how developers perceive ASATs, specifically which barriers they face in the adoption of these tools and how these tools should be created so that developers will take advantage of them. For this purpose they surveyed a random sample of 375 Microsoft developers. What was considered to be the largest obstacle in using these tools was the fact that some unwanted rules are turned on by default in the tools' configurations. Proposed solutions to this problem are to have a subset of rules enabled instead of all rules being enabled by default, or to make the configuration process easy for developers. This idea is similar to what ESLint does; there are no default configurations but one can easily enable a preset with a selected subset of rules that the creators of ESLint think are the most important (`eslint:recommended`). Other frequently experienced challenges were bad warning messages, too many false positives and for the analysis to be too slow. For the functionality that these developers want ASATs to detect, security issues, best practices and concurrency issues were the most commonly requested ones. Issues relating to style were among the least requested features for such a tool to have. However, the majority of the developers reported that the issues they encounter that can be caught by ASATs are most commonly best practices violations and style inconsistencies, opposed to more complex issues such as regarding reliability or concurrency. The expectations of these developers therefore seem to match with a tool like ESLint.

## 3.2 Configurations of Static Analysis Tools

Ayewah *et al.* [64, 65] studied the usage of FindBugs where they saw that users are generally interested in fixing warnings from the tool, especially the high priority ones, but which types of warnings depends on the user's context. It is therefore deemed valuable to have configuration options for these different groups of users. Similarly, Jaspan *et al.* [105] recognized the importance of customizing and prioritizing rules to make developers more willing to use an ASAT as it reduces the number of perceived false positives for a project.

Regarding how developers configure these tools, Beller *et al.* [69] performed a large-scale study on the prevalence of ASAT usage along with how they are configured. They examined ASATs for four languages, Java, JavaScript, Python and Ruby, where for JavaScript they observed JSHint, ESLint, JSCS and JSL [34]. They found that these tools are commonly used in OSS software (over half of all analyzed projects) and in particular for JavaScript projects where JSHint was by far the most widely used linter[1]. The configurations for these projects are most often changed from the default settings, but typically only one rule is added, removed or modified. Furthermore, after a configuration file has been created, they are rarely modified. For the types of rules that are configured, those that are related to maintainability were both more commonly enabled and disabled than those that have to do with functional defects. They predict that fewer bug-finding rules are enabled as tools are known to perform poorly in finding defects.

Zampetti *et al.* [168] further analyzed the usage of ASATs in continuous integration (CI) pipelines in popular OSS projects. For the 20 analyzed projects, Checkstyle was the most commonly used ASAT (13), followed by FindBugs (11), where 11 projects used only one tool while the rest employed multiple tools. Most of these ASATs were configured to break the build in the CI pipeline, especially in the case of Checkstyle (only one project solely raised warnings). Most projects (17) manually configured the rules that were activated in these projects for at least one tool that was used, where Checkstyle was configured

[1]At the time of analysis in early 2015.

in every single case. While FindBugs was typically configured to include all available rules, the percentage of used rules for Checkstyle always remained below 40%. The most commonly enabled Checkstyle rules were regarding indentation (*Whitespace*), unused imports (*Imports*), possible defects (*Coding*) and code blocks (*Block checks*). The configurations of the ASATs were not changed often over the projects' lifetime, in all cases except one less than 10 times. The percentage of builds that were broken because of the ASATs was often 0% and typically below 10%, as developers are likely to be careful to not break the build by first running the tools locally or by being careful in writing the code. For warned builds, the percentage was above 10% in each case and sometimes up to almost 100%, showing that warnings often remain unfixed in the code. Maintainability issues more commonly caused builds to fail rather than possible defects in code. When builds fail due to ASAT errors, they are typically fixed quickly or within 8 hours, by fixing the issue in the source code. Zampetti *et al.* recommend developers to configure ASATs to their needs to benefit the most from using these tools, *e.g.*, to avoid unnecessary build failures.

## 3.3 Effectiveness of Static Analysis Tools

Several studies have focused on the effectiveness of ASATs to find bugs in software systems. In a case study, Wagner *et al.* [163] found that ASATs report on different warnings than those that are found by testing a software, indicating that it is beneficial to use these two methods in combination. Furthermore, ASATs find a subset of the issues that are found via code review, thus if used beforehand, they can some of the time that goes into performing the manual task. Zheng *et al.* [169] also found that ASATs are complementary to other fault-detection techniques but with a limited usability where testing was found to be more effective.

Wedyan *et al.* [164] further examined how effective ASATs are in detecting faults and refactoring opportunities. Studying two software systems, three known ASATs were only able to detect 3% of the defects that were actually found in the projects' lifetime. These tools were however much more successful in identifying the refactorings that were performed in the project, or around 70%. Similarly, Couto *et al.* [77] also found that FindBugs was not able to identify most of the errors that had been fixed in some studied software systems. However they found some level of correlation between the number of reported warnings and the number of actual field bugs. So despite the fact that ASATs may not be good at finding actual field defects, they can serve as good indicators of the level of internal quality in a software system.

## 3.4 False Positives in Static Analysis Tools

A common problem with static analysis tools is the high volume of false positives [164, 108, 71, 100]. For example, Wedyan *et al.* [164] found well known ASATs to produce a false positive rate above 90% when applying them on several software systems. The presence of false positives is indeed known to be one of the main barriers of ASAT adoption [108, 74, 139]. Developers will not tolerate a high false positive rate and quickly discard a tool if the rate is too high [74, 141]. More specifically, Christakis and Bird [74] found that most developers do not tolerate a false positive rate of over 20%. They prefer for tools to identify fewer defects, rather than catching more bugs and having a higher volume of false positives. Furthermore, Wagner *et al.* [163] experienced that when ASATs report a high number of false positives, it can take even more time to shift through them to find the true defects, than the time that is saved on manual effort by using an ASAT in the first place.

The term false positive can be understood in two different ways, either a wrongly reported warning or a true warning that is not considered by a developer to improve the software under analysis [67]. Tool makers have thus experienced that the users decide for themselves what a false positive is and thus how effective a tool is [141]. Additionally, when tools identify more intricate issues, the risk increases of users not understanding the issue and mistaking true positives for false positives [71]. In general when there are many reported false positives, developers start trusting the tool less and less [71]. With less trust, even more true positives are thought to be false positives by the developers. Couto *et al.* [77] and Jaspan *et al.* [105] further experienced that developers categorize warnings that are not of their interest as false positives. They stressed the importance of customizing tools and prioritizing rules so that only high-priority and relevant warnings are reported. Without doing so, tools can not become practically usable.

## 3.5 Static Analysis Tools Used in Industry

An abundance of ASATs have been developed with diverse features and implementations, where software systems have different requirements for these tools. Many ASATs have been used by major software companies such as Google, Facebook and Microsoft, both tools that have been created internally and externally [74]. Bessay *et al.* [71], the creators of Coverity, experienced first hand how difficult it can be to create an ASAT that fits for commercial use of companies with large code bases. The technical requirements of such companies can vary greatly, *e.g.*, using miscellaneous build processes and developing environments that can make running the tool different for each particular case. The size and nature of software projects can also entail different requirements for ASATs, where Google resorted to creating their own tool as no other available ASAT could handle the size of their codebase [141].

Google have indeed tried and reported from using several ASATs such as FindBugs and Coverity [66]. Despite some success with adopting these tools, ultimately they were not frequently used due to the presence of false positives, lack of scalability and inconvenient workflow integration [141]. Eventually they instead created their own tool, Tricorder [141], an ASAT with an ecosystem of developers creating and reviewing custom analyzers. They only display high priority warnings from analyzers that have received good reviews from other developers, in an attempt to make the developers more willing to use the tool by eliminating false positives and low priority warnings.

EBay [105] reported on their substantial effort into choosing the most valuable and appropriate tool set for their projects. They report on a method to evaluate these tools where eventually the tool FindBugs was integrated to their development process. For their software it was deemed important that the adopted tool should be extensible (to allow for project custom rules) and customizable. They report that customization and prioritization are important steps in evaluating a tool as it can reduce the false positive rate for a project and make developers more willing to use the tool.

## 3.6 Static and Dynamic Analysis for JavaScript

As JavaScript has become more popular it has also attracted more attention in the research community. The need for more and better tooling for the language has been emphasized along with the challenges that JavaScript introduces for proper static analysis. Several studies have therefore focused on techniques for, and the creation of, static analysis tools for JavaScript [107, 146, 126].

As discussed in Chapter 2, JavaScript is a dynamic language, which *e.g.*, involves dynamic typing, dynamic file loading, first-class functions and property access, and in general code that can be generated during runtime. Richards *et al.* [138] analyzed the usage of dynamic features in JavaScript code in commonly used websites and found that some dynamic features are indeed frequently used, such as adding and removing properties to objects after their initialization and using the `eval` function for various unpredictable purposes. As such, JavaScript has been described as harsh terrain for static analysis [138, 126, 106, 142]. The usability of ASATs for JavaScript has thus also been criticized and said to be limited since these tools can not account for runtime behavior [94]. For this problem, tools have been created that employ dynamic analysis of program executions [142, 94, 86]. A recent example is DLint [94] that, evaluated against JSHint, detected on average 49 new code quality issues per studied system. Example issues they identified are object properties shadowing prototype properties, passing too many arguments to a function, accessing the `undefined` property, and silent coercion of arithmetic operations to `NaN`. Some issues were detected by both DLint and JSHint but with different methods to do so, where DLint increased the total warnings found by 10%, thus complementing JSHint.

In general, while there have been several proposed static analysis tools and techniques for JavaScript in previous research, most suffer from being *sound* but impractical, or the opposite, practical but *unsound* [119, 142]. Practical limitations include lack of scalability and inabilities to analyze library usage. More tools have been developed for specific challenges of JavaScript, such as the prevalent use of external libraries [63, 126]. Some large libraries, such as the Browser API and HTML DOM, are written in other languages and do not provide a JavaScript implementation, thus making static analysis more difficult. Other common libraries like jQuery are also known to make extensive use of many dynamic language features, such as use of `eval` and computed property names, which also contributes to the difficulty of static analysis for JavaScript code. Another related challenge is the dynamic file loading of these libraries where the code is therefore not made available immediately for analysis. For this problem, researchers have proposed *staged* analysis [76, 97] which is applied in two stages, first for all statically available code and later for the remaining dynamically loaded code. Interestingly, the false positives that have been identified with static analysis of JavaScript code (by applying the tool SAFE [123]), mostly had to do with API usage and asynchronous function calls, though also due to dynamic file loading and dynamic code generation [132].

Despite the challenges that it brings, automated static analysis can be especially beneficial for languages that do not have strict typing [169]. Type analysis is considered to be a crucial part to catch representation errors where, for example, numbers can be confused with strings or functions confused with booleans [107, 133]. Additionally, simple mistakes such as spelling mistakes can result in surprising consequences at run time, which is mostly avoided in other languages with static typing [106]. Pradel *et al.* [133] created a tool, Type-Devil, to counter against errors that have to do with inconsistent types in JavaScript, using dynamic analysis where types of variables, properties and functions are observed at runtime. Moreover, as typing systems have been developed such as TypeScript [72] , Gao *et al.* [89] studied the effects of dynamic typing on errors in JavaScript applications and examined whether typing systems could prevent them. They added type annotations to code that had caused errors in software systems and examined whether using TypeScript or Flow on that code would surface the previously encountered errors. They saw that applying these static type systems would detect many errors (15% in this study) that otherwise can go unnoticed into production code.

Lastly, code smells in JavaScript have recently been studied where Fard *et al.* [86] developed a tool, JSNose, to detect potential smells. Saboury *et al.* [140] later analyzed the prevalence and impact of several code smells and surveyed developers on their perceptions of those smells. The studied code smells were derived from literature and online style guides, including recommendations from ESLint and from the Airbnb preset. They found that JavaScript files that are affected with code smells tend to be more error-prone than those without any smells. Moreover, *Variable Re-assign* and *Assignment in Conditional Statements* were among the most dangerous smells. Additionally, when surveying a large sample of JavaScript developers, *Nested Callbacks*, *Variable Re-assign* and *Long Parameter List* were perceived to be the most hazardous smells affecting the maintainability and reliability of JavaScript applications.

# Chapter 4

# Developers' Perceptions of Linters

*This chapter was conditionally accepted as a paper in the proceedings of Automated Software Engineering (ASE) in 2017 [159].*

For this chapter, we aimed at understanding how and why developers use JavaScript linters in real world applications. More specifically, we investigate the motivation behind using a linter, namely ESLint, and what methods developers apply to configure the tool. To that goal, we used a qualitative research method inspired by Grounded Theory [93, 62] to conduct and analyze interviews with 15 developers from reputable open source software (OSS) projects. By doing so, we obtain valuable insight into real world usage of these tools and increase our understanding on how they can be used. Our main results from this study consist of in-depth knowledge and actual use cases that can benefit other developers, tool makers and researchers. We describe the different benefits that JavaScript developers obtain when using a linter and with that demonstrate to other developers why using a linter can be valuable for them, as well as helping tool makers to understand the important parts of such tools according to their users. Furthermore, we gather various methods regarding how developers decide on which rules to include in configurations for a project, thus helping other developers with using a linter and creating a suitable configuration for their projects. Lastly, we collect several challenges that developers face when using a linter which helps tool makers to improve future versions of linters and provides researchers with opportunities for further research.

## 4.1 Research Questions

Following the main ideas of Grounded Theory, the purpose of these interviews was not to test any detailed preconceived hypotheses or research questions, but rather to let the different views and ideas emerge. Nonetheless, general objectives were established in order to perform somewhat structured and efficient interviews. More specifically, the goal of the interviews was to examine:

**RQ$_1$ Why do developers use linters?**

**RQ$_2$ How are rules selected for a project?**

**RQ$_3$ What are the challenges in using a linter?**

While we do post these as research questions, their intention was more in the direction of guiding the interviews. We therefore do not post any more detailed questions here, as will be done in later chapters.

## 4.2 Methodology

We followed a qualitative research approach [78], inspired by many concepts of Grounded Theory [93, 62] where the aim is to discover new ideas emerging from data instead of testing any preconceived research questions. With an open mind we wanted to understand why and how developers use a linter for JavaScript. For that purpose we collected data by conducting 15 interviews with developers from reputable JavaScript projects on GitHub. In the following sections we explain the process of designing and conducting these interviews, along with how the data was analyzed.

### 4.2.1 Interview Design

The first task was to design the interview and decide on which questions to ask. The first questions were written to be best aligned with the previously stated research goals. Further inspiration for questions was received by browsing through ESLint configuration files of popular JavaScript projects on GitHub where some common patterns were noticed but also the diversity of the different configurations. This was an iterative process where many possible questions were constructed which were finally narrowed down to a shorter list of 13 questions. It was intended to keep the interview as short as possible (*e.g.*, around 20 minutes) to make it more appealing for developers to participate in them, but still for it to be possible to extract valuable information.

Each interview was designed to start off with broad questions in order to open up a discussion early where the participant could express his or her experience with linters in a rather free and hopefully honest way, without being directed to any particular path. These questions include *Why do you use a linter in your project?* and *How do you create your configuration file and maintain it?*. More specific questions were then asked later on in the interviews, such as *Do you experience false positives? if so, which?*. The complete list of questions is available in Appendix A.

Some helper material was created to make it easier for the participants to answer some of the more challenging questions. More specifically, we wanted to know which individual rules the participants consider to be the most important ones, but soon realized that coming up with an answer to this might be difficult. Participants might not have a preconceived answer to this question and as there are 236 available rules from ESLint, developers might not recognize all of them and going through each and every one in the interview would take too much time. To make this question a realistic possibility, it was attempted to improve the visualization of looking through the available rules. A document was therefore created for each participant where the available rules were split up in a separate table for each of the seven ESLint categories. Each table had two columns, one for rules that were in use by the participant's project and the other for the rules that were not in use. Furthermore, the rules were colored based on their setting: red for rules that were enabled as errors, orange for rules that were enabled as warnings and green for rules that were turned off. Moreover, the rules that are included in the recommended settings by ESLint were written in bold. This setup was intended to make it easier for the participants to identify which rules they might think are important, based on their presence in the project's configuration files and in the recommended settings by ESLint. An example of this visualization can be found online [155]. On a similar note, we wanted to know which rules were the least important ones, which would most likely be an even more difficult question for the participants. For this purpose it was decided to simply ask instead if there were any specific reasons for not choosing some rules.

### 4.2.2 Interview Procedure

The interviews were conducted in a semi-structured fashion as it is commonly done in software engineering research [99]. With this method, specific questions are combined with open-ended questions to also allow for unexpected information in responses [143]. Hove and Anda [99] describe their experience of performing interviews in software engineering research and encourage interviewers to talk freely, to ask relevant and insightful questions and to follow up and explore interesting topics. These guidelines were followed while conducting the interviews. The interviews were dynamic in nature where follow up questions were asked based on the participants' replies, often resulting in interesting discussions.

To facilitate the interviews, participants were asked to take part in an online video or audio call on Google Hangouts [32]. These sessions lasted from 16 to 60 minutes, with an average duration of 35 minutes. Each interview began with a quick introduction of the project and of the author, after which the participants were asked for their permission for the call to be recorded and for them to be possibly quoted in the thesis or in a published article (for which all replied positively). The interview itself could then begin as shown in Appendix A, where the participants were also provided with the previously mentioned helper material. Three out of the 15 interviewees were not able to participate in an online call and instead received a list of questions via email and gave written responses.

### 4.2.3 Data Analysis

Continuously after each interview, *memoing* was done to note down ideas and to identify possible categories in the data. The interview recordings were then ultimately manually transcribed. As parts of the interviews included casual chat about the topic, some irrelevant information along with repetitions were left out and some parts were summarized.

First, *open coding* was performed where the transcripts were broken up into parts that related to the topic of each question in the interviews, also including the different follow-up questions. Secondly, *selective coding* was done where more detailed categories were identified within the three main themes, which then became the more detailed topics that are presented later as results. In this process we took advantage of the memos that had been written over the course of conducting the interviews. Too keep the coding document efficient and organized, direct quotes were not included but only references to the interviewees. It was then easy to look up the relevant quote in the document that was first discussed. The complete list of codes and the corresponding references can be found online [151].

### 4.2.4 Participants

In order to find potential participants for the interviews we examined the most popular JavaScript projects on GitHub, according to their number of stars in December, 2016[1]. It was considered that by observing the top projects on GitHub, one can obtain an insight into active and reputable projects with many contributors, providing more interesting and relevant information about the usage of linters. We detected those who 1) use ESLint, 2) have some custom configurations and 3) where one or two contributors could be identified that have been more involved than others in editing the configuration file. The second criteria was used to find respondents that would have more to say than perhaps to only shortly explain why they use one particular preset. An email was then sent to one or two main contributors of the ESLint configuration file of the corresponding project, asking for a short interview. This email was carefully written to be as concise as possible but also friendly and inviting, in an attempt to make the receiver more likely to read the content and respond positively. The purpose of the email was explicitly stated in the beginning, followed with a short description of the interview purpose and some information about the author, and again ending with a request to participate in a short interview. These requests were sent out in batches of 5-10 emails (starting with the most popular projects) as it was difficult to predict how many positive replies we would receive. Batches were sent out until we had received a sufficient number of positive replies back, where the goal was to perform at least 10 interviews, or until we were satisfied with the amount of information we had collected.

A number of 120 projects were eventually examined where 37 requests were sent out. These resulted in 15 interviews being performed, thus with a response rate of 40%. The information from these 15 interviews was considered enough to provide us with *theoretical saturation* [93]. Table 4.1 shows the developers who participated in the interviews where, in order to keep the participants' anonymity, they are given names starting with the letter P and followed with a number from 1 to 15. The number of months for which each corresponding project had used ESLint is also displayed[2], where most projects had migrated from using another linter such as JSHint. The table furthermore shows the placement of the projects in the top 120 JavaScript projects on GitHub within a range of 10 projects each, also to maintain the participants' anonymity. A summary of the participants' experience is shown in Table 4.2 where the average experience as a professional software developer was

[1]The most popular projects are ordered by the number of stars they receive from other GitHub users. The purpose of starring a project is to keep track of projects that a user finds interesting or simply to show appreciation to the repository [90]. The most starred JavaScript projects on GitHub can be found online [92].

[2]As of February 2017.

| Code | Months | Placement |
|---|---|---|
| P1 | 25 | 11-20 |
| P2 | 22 | 11-20 |
| P3 | 5 | 21-30 |
| P4 | 14 | 21-30 |
| P5 | 8 | 31-40 |
| P6 | 7 | 41-50 |
| P7 | 1 | 61-70 |
| P8 | 23 | 71-80 |
| P9 | 5 | 81-90 |
| P10 | 3 | 81-90 |
| P11 | 4 | 91-100 |
| P12 | 16 | 91-100 |
| P13 | 15 | 111-120 |
| P14 | 24 | 111-120 |
| P15 | 22 | 111-120 |

Table 4.1: All participants' codename, number of months using ESLint in the corresponding OSS project and the range for the project placement in the top 120 JavaScript projects on GitHub

| | Low | High | Average |
|---|---|---|---|
| **Years as developer** | 3.5 | 27 | 11.8 |
| **Years as JS developer** | 1.3 | 20 | 8.9 |
| **Years in project** | 0.6 | 5 | 2.7 |
| **Project age** | 1 | 8 | 5.1 |

Table 4.2: Experience of participants, showing the lowest and highest answers along with the average of all answers

.

11.8 years. Among the 15 participants, four are founders of the project, seven identified themselves as lead or core developers and four are project maintainers.

### 4.2.5 Limitations

In the following we address the possible limitations of this analysis.

#### 4.2.5.1 Interview Design

We attempted to make a particularly complex question to be easier to answer by providing the participants with a different visualization of the available rules from ESLint. The ordering and emphasizing of the rules in this document could possibly have affected their answers to this question more than intended, where they might have stressed more on those rules that they noticed first, thus possibly threatening the *construct validity* of this analysis. This visualization method was however considered to be crucial to be able to ask this particular question and receive accurate answers in a short time. However ultimately, this part of the interviews did not become a key part of the analysis.

#### 4.2.5.2 Interview Procedure

While most interviews were performed via live video or audio calls, three out of the 15 participants answered the questions in writing. The nature of the interviews was rather different in these cases where no further discussions could emerge as no follow-up questions could be asked. The written answers were generally much shorter than the verbal answers, although concise and to the point. While this makes the data somewhat different, it is not considered to compromise the quality of the answers. They were not as elaborate as the verbal ones but accurate enough to also provide valuable information.

Both when asking the developers via email to participate in the interview and when starting each interview, the goal and purpose of the research was described. It is therefore possible that the participants were biased to express either a positive or negative experience after being told this information, perhaps resulting in some level of *hypothesis guessing* [166]. It was attempted to minimize the risk of this by providing as little information beforehand as possible.

#### 4.2.5.3 Data Analysis

The coding of qualitative data can be a delicate process. In this case it was conducted by one person which can involve risks of *experimenter expectancies* [166]. It is possible that someone else would have coded the data differently and perhaps resulting in different conclusions. This was mitigated to some extent by focusing on not having any preconceived ideas before processing the data and thus not trying to fit the data to any existing theories. Furthermore, the derived codes are available online for inspection [151].

#### 4.2.5.4 Participants

The *external validity* is threatened by the sample size of participants as it is not large and thus may not represent all OSS development. Moreover, as we only talk to developers from OSS projects, the results may not represent industry software. We tried to mitigate this fact by interviewing experienced developers from popular and reputable projects that furthermore have expertise on the topic of using linters for JavaScript. However, that selection of the sample creates another bias in the study where the results might be different if smaller projects were examined along with projects having fewer configurations. Only projects were selected that had somewhat extensive configurations as they would have more input on how rules are selected for a project, thus being able to report on what methods they use for the task. These findings therefore might not reflect smaller projects with minimal configurations. In the next chapter we examine the configurations of a much larger sample which includes JavaScript projects of all sizes. Then in Chapter 6 we survey another large sample of JavaScript developers where we challenge the findings from this analysis. The final results and conclusions are based on all three analyses, thus minimizing the effect of this sampling bias.

Regarding the *internal validity* of this chapter, possible variables that effect the results relate to the previous knowledge of the participants. It is likely that we interviewed people who already feel strongly about linters as they are frequent users of the tool. We can not know for sure if their opinions are based on their own experiences with using the tool or if it is based on external information such as literature that they have read. Because of this concern, we tried to address the questions to relate specifically to the participants' own opinions and experience with working on the particular project. However, in some cases, the participants had other and even more experience in working with a linter on other projects, for which they also based their answers on.

## 4.3 Results

In the following we present our results on why and how JavaScript developers use linters, along with the challenges that they face.

### 4.3.1 $RQ_1$ Reasons for Using a Linter

Multiple reasons were reported by the participants as to why they use a linter, which are discussed in the following sections.

#### 4.3.1.1 Prevent Errors

Using a dynamic language such as JavaScript is not free of risks: "*Without a linter [JavaScript] is a very dangerous language. It's very easy to make a very big problem and then spend 30 minutes to find it.*" (P7). The majority of the participants reported that the number one reason as to why they use a linter is to catch possible errors in their code: "*There are things which are easy mistakes to make and are obvious errors and I think those provide the highest value. Because you have a one to one correspondence between times that a rule catches something and bugs that you've prevented.*" (P4). More explicitly, when asked about the most important category of warnings in ESLint, 10 participants answered that Possible Errors was the most important one (P1, P2, P3, P4, P5, P7, P8, P9, P11, P12): "*Possible Errors is the #1 most important, the biggest reason to use a linter is to catch errors the programmer missed, before they become a runtime bug.*" (P9). So not only does it catch many bugs but it also does so early in the development process: "*If you can get some bugs away from your code so early as when you write it, it's great.*" (P12).

These rules can also be especially useful for bugs that are hard to find and to debug: "*It is nice that you have that kind of safety net because often times these are things that the average developers are aware of and if you do hit those cases it could be a nightmare to debug.*" (P15). An example of this situation is using two equal marks when three should have been applied, which can cause substantial unpredicted problems but the cause can be very difficult to find (P8).

A special category of bugs in JavaScript has to do with the declaration of variables because of the dynamic nature of the language. Indeed, there is a specific category in ESLint that only contains rules that have to do with variable declarations and errors regarding variables (appropriately named Variables): "*Possible Errors will catch a lot of unintended behavior, and Variables will catch a good deal more.*" (P5). Two participants reported that Variables was the most important category (P10, P15). For example, when a developer mistypes a variable or uses the wrong variable name, the linter can catch it and warn the developer: "*It's very easy to write JavaScript code that has errors, you might use a variable that hasn't been declared or you might have a typo in your variable name and because JavaScript is often not compiled, you'll only discover that much later when you run the code.*" (P1). More specifically, nine participants (P1, P3, P4, P5, P8, P10, P12, P13, P15) mentioned the importance of the rule `no-unused-vars` (identifies variables that have been declared but never used) and five (P1, P3, P5, P13, P15) mentioned `no-undef` (identifies variables that have not been defined) which are both useful to detect mistyped variables. Other examples of errors that can be avoided with ESLint is detecting duplicate keys in objects (`no-dupe-keys`) and duplicate names in function parameters (`no-dupe-args`) (P3, P4, P8, P15). The presence of these duplicates can cause unexpected behavior in an application: "*When I added ESLint to my project, these rules found real problems in my code.*" (P8).

An interesting case was when a participant originally started contributing to the OSS project solely to make ESLint prevent a specific bug from occurring. More specifically, participant P13 noticed that there was a bug in the project that caused his test suite in another project to crash. He knew that this bug could be avoided by applying a certain rule from ESLint. Therefore, to make sure that this particular error would not occur again, he started to contribute to the project by enabling ESLint and by creating a configuration file where this rule was included.

#### 4.3.1.2 Augment Test Suites

While linters are being used to catch errors in code, there is another popular and widely accepted method to catch bugs which is to write unit tests. It is therefore interesting to know how these two methods are combined for this purpose. Some participants mentioned that they use a linter on top of unit testing as a complementary approach to the regular tests (P1, P3, P8). P1 and P8 pointed out that unit tests commonly do not cover all code which can result in problems being easily missed: "*You need to seek all possible cases for unit tests, but sometimes it's very hard, and of course in all projects, unit tests don't cover all possible cases. So this is why a linter is a second protection line.*" (P8). In some cases a developer might have written tests for new code but then makes a final refactoring or clean up and forgets to update the tests as well: "*Definitely I've done that before and the linter has helped me catch those errors beforehand.*" (P3). Furthermore, the tests can also take substantial time to run and thus the linter can be seen as a much faster version of smaller subtests (P1).

On the other hand, participant P4 believes that unit tests and manual tests can usually cover all errors, so even though ESLint would not be used, the errors would eventually be caught by the various tests that are applied. However he says that the linter can catch them earlier in the process and is also better at identifying code that is ambiguous.

#### 4.3.1.3 Avoid Ambiguous and Complex Code

It can be difficult to understand code correctly where the intention is not perfectly clear. The category Best Practices tries to tackle this problem where, according to its documentation, it contains rules that relate to better ways of doing things to help developers avoid problems. While only one participant recognized the category to be the most important one (P6), others identified it as the second most important (P4, P8, P13, P15). Some of these rules try to prevent code from being misunderstood: "*It helps enforce code which says what it does, so that it's easy to understand.*" (P4). In some cases code is actually doing something else than it appears to and a linter can help to detect these situations (P2, P4, P6). One example of this is restricting the usage of switch statements by forbidding the use of "fall throughs"[3]. This is done because the intention of switch statements can be easily misunderstood when that feature is used (P4). Another example is where the linter identifies code that is unreachable (rule `no-unreachable`). According to P2, not only can it catch possibles mistakes by a developer but it can also help with removing a lot of unnecessary code that otherwise makes the codebase more difficult to understand and where the intent of the code can be unclear. Moreover, four other participants mentioned that this is indeed an important rule to use (P1, P3, P6, P15): "*Having code in your app that's never going to run sounds like the worst idea ever.*" (P1).

#### 4.3.1.4 Maintain Code Consistency

Every single participant mentioned that one of the reasons why they use a linter is to maintain code consistency. Having a consistent code style in a project is beneficial for many reasons, one being that it improves the readability and understandability of the code. As an example, P10 reported that inconsistent code, such as having different spaces and semicolons, makes the code very difficult to read and understand since in those cases these inconsistencies consume all his attention. This might even be especially relevant in the case of JavaScript since it is a language where the developer has substantial freedom in how to write the code (P12, P14): "*With JavaScript you can write code in many ways, and it can be hard to read other people's code if you write it in a different way.*" (P12). This problem was also compared to the broken windows theory [165] which says that if a window in a building is left broken, all other windows will soon be broken as well: "*There's a huge amount of value in having consistency in code, it's the whole broken windows thing, people take a higher quality care of the code if they're expected to meet a certain level of consistency, from my experience at least.*" (P6). The theory further explains that these broken windows can then lead to more small crimes which can eventually lead to more serious crimes. Maintaining rules of a consistent code style could therefore lead to better overall quality of the code.

[3]A "fall through" occurs when all statements after a matching case are executed until a break statement is reached, regardless if the subsequent cases match the expression or not.

This topic relates mostly to the category Stylistic Issues where there are many different rules available to enforce specific code styles. Even though every single participant mentioned this matter in the interviews, it does not seem to be of high priority for them. When participants were asked which category of rules they thought were the most and least important ones, two considered Stylistic Issues to be the most important category while 10 thought it to be the least important one. Some participants were bothered by the fact that choosing which style to follow is a very subjective decision and developers generally have very different opinions on how code should be written (P1, P3, P5): "*Stylistic Issues - they're all opinion based.*" (P5). On the other hand, four participants explained that Stylistic Issues was indeed the least important category simply because other categories can catch bugs which is far more important (P2, P4, P9, P13). This category thus still provides a lot of value and they would not want to omit it: "*They make [the code] harder to read but they just don't cause issues as much.*" (P13).

#### 4.3.1.5 Faster Code Review

In order to uphold code consistency, project maintainers can monitor and review new code that is proposed. GitHub projects commonly make use of pull requests to submit new code where other developers can review the changes and write comments on them. There are several aspects to consider when reviewing pull requests, such as functionality and code style. Several participants mentioned that they use the linter to avoid having to manually review the code style in pull requests (P1, P2, P3, P4, P8, P11, P14, P15): "*Any software project wants to maintain some bar of quality and many of the ways that we assess that are going to need human intervention, but there are a subset of those problems that a piece of software has which can be done by a computer and as engineers I think we are apt to try to use computers to solve human problems where possible. You can free up human time to do more interesting things.*" (P4). The time that would be spent on reviewing the code style can be long and developers would like to use that time for other more important things: "*You spend a lot of time to fix code style problems and this is the second reason why I love linter tools, because they make this work for me.*" (P8). Furthermore, it saves time for the contributor of the pull request since he or she receives much faster feedback from a linter than from a person that would conduct a review (P4).

Maintaining code consistency with a linter can also make pull requests much easier to review. When there is a set of stylistic rules in a project to which everyone has to conform, all pull requests have minimal stylistic changes. If there are no rules, there can be multiple code changes of *e.g.*, only whitespaces or line formatting which might be caused by different editors being used. This can make it difficult to see the actual changes that were done in the contribution since they are hidden by these formatting changes (P3, P12).

#### 4.3.1.6 Spare Developers' Feelings

Developers can sometimes be sensitive to criticism when receiving comments during a code review (P2, P8, P11). This can particularly be the case for new developers: "*If you tell to a new developer that he or she made a mistake, it will be very sensitive. He may feel very* *uncomfortable because somebody found a mistake in his work. But if a linter tells you about a mistake, it is not a problem, because it's not personal.*" (P8). A new developer might also look up to the person that is conducting the code review which can make the criticism especially dispiriting (P2). Having a linter doing this job can also contribute to people feeling more equal in a project if there is no senior person telling others to do things differently (P11).

#### 4.3.1.7 Save Discussion Time

Having a set of rules regarding code style can also save time that is spent on discussing different styles and opinions (P2, P4, P5, P6, P7, P10). In big projects with many contributors there can be many pull requests in circuit and discussions can occur where developers disagree on a certain style that is used. P2 explains that discussing code styles is not worth the effort when there are other more important things to discuss. He further describes that comments regarding code style on pull requests can be different depending on which developer is conducting the review. In some cases, contributors can therefore receive contradictory advice if no rules exist that everyone goes by.

The discussions about code style that can occur in pull requests or in issues can also even lead to arguments between people since developers have very different opinions on the matter (P2, P3, P5, P7). All this can be avoided by deciding upon a set of rules to begin with: "*It's almost like a code contract. There may be things that each of you have assumed and you don't know what your own assumptions are, and what could possibly lead to conflict down the road, so you have a written contract to try to address everything up front.*" (P7).

#### 4.3.1.8 Learn About JavaScript

ESLint can be used to learn about new JavaScript features or new syntax. P12 used ESLint to help him to learn the new syntax of ECMAScript 6 (ES6): "*When I switched to ES6, I used it as an educational tool. It's so easy to continue to use var for variable declarations. I used ESLint very strictly to enforce ES6 syntax, otherwise I would probably still use ES5 when I write code. But with the help of the linter it forces you to switch to ES6, which is a good habit.*" (P12). In a similar fashion, he used a custom rule set for React [49] (a JavaScript library for building user interfaces) when using the library for the first time. He was notified by ESLint how the React code could be written in a better way, not just regarding formatting but also so that it would be better for execution. With that he learned how to write both correct and efficient React code.

Even though linters can be beneficial to all JavaScript developers, they can be especially helpful for new developers, either those who are new to a project or those who are new to programming in general (P6, P7, P9, P13, P14). Contributors in OSS projects usually have different levels of experience and using a linter can help with "*leveling the playing field and helping people to understand what's actually going on*" (P13). This particular example came from a developer that had been working with students who were accustomed to getting errors from the Java compiler, telling them what they can and can not do. However when using JavaScript, one can run code that includes various coding mistakes and not get notified about it (P13).

Some features of JavaScript can also be used in clever ways with enough knowledge of the language. However, P7 explains that, without sufficient knowledge, those features can be used in the wrong way and lead to unwanted behavior. For example, experienced developers can use a double equality with perfect intentions, but a newcomer in JavaScript might use it by accident, not knowing that a triple equality should have been used instead (P7). ESLint indeed has a rule that warns developers about the usage of two equal marks (rule `eqeqeq`).

Having non ambiguous code and a consistent style can also be even more helpful for new developers (P6, P9). Having consistent code can make it easier for new developers to read the codebase and get up to speed quickly. Using a linter "*makes sure it's the same throughout so that when other people come in and want to add a feature, they're not really confused for what they should be doing. It's easy to follow.*" (P6).

**RQ$_1$ Why do developers use linters?**
The most common reasons for using a linter are to maintain code consistency and to prevent errors. Other reasons for using a linter are to augment test suites, avoid ambiguous and complex code, perform faster code reviews, spare developers' feelings during code reviews, save time on discussing code styles and to learn about the JavaScript language.

### 4.3.2 RQ$_2$ Configuring Linters

In the following we present the various methods that were reported for which developers use to configure linters.

#### 4.3.2.1 Use an Existing Preset

There are many publicly available presets that anyone can use in a project instead of creating a custom configuration, or use as a part of a custom configuration. These presets have been carefully constructed by their creators and have been changed attentively over time: "*They thought about the code standard quite extensively and put a lot of thought in it.*" (P12). Several participants like to use a preset as a part of their configuration file (P1, P6, P10, P12, P13, P15) and one normally tries to solely use the preset as is (P8).

One specific reason for using a preset was mentioned by P13 where he does so to hide the settings from other developers so that they will not change them: "*Developers will try to change the settings if they disagree with anything, so using [ESLint] recommended or a preset stylesheet prevents that.*".

#### 4.3.2.2 Project Fit

It is important that the stylistic and best practice rules fit the existing code when the rules are chosen (P3, P4, P6, P12): "*I wanted them to fit the code as it was, I wanted the linting in* *place with as little mess as possible.*" (P12). P4 explained that if something is already being done in a project, it is rather straightforward to enable a rule that ensures that it will continue to be done the same way in the future. Furthermore, if there is already some sense of style in the existing code, it is not very sensible to change it to something else since it would create more work than necessary when setting up a linter. As an example, P4 mentioned that if a preset contains a rule that enforces either spaces or tabs and the setting is opposite to what has commonly been done in the project, the particular rule will be overwritten to fit better to the project. Otherwise, multiple stylistic changes would need to be done to conform to the rule.

#### 4.3.2.3 Automatically Generated

Extending the last approach, ESLint provides an automatic method to insure that a configuration fits well to a specific project. The source code is inspected to detect a common code style and the user is asked a few questions, and from this a suitable configuration is created. Two interviewees used this method to create their configuration file (P11, P14). P11 then looked over the generated rules and the errors in the output to see if he agreed with them. He does not consider it to be wise to use a preset since linter configurations are generally very project dependent: "*I didn't even consider Airbnb or Google because I think every project is a little different.*" (P11).

#### 4.3.2.4 Pull Request Discussions

Three participants (P2, P4, P10) reported that when something is discussed in a pull request that can be enforced with a rule, they use the opportunity to enable the corresponding rule. Since the topic surfaced in the pull request then there was obviously a need to make a decision on the matter being discussed, whether it is a code style or a best practice. Furthermore, that way the topic will not surface again and time will not be spent on discussing it, as it will be already decided on in the configuration file (P2).

#### 4.3.2.5 Minimal Configurations

Some prefer to keep the configurations as simple and minimalistic as possible (P1, P5, P8, P14, P15): "*We don't want people to feel like they have to jump through unnecessary hoops to get their PR's in, so turning on every single thing wouldn't be great.*" (P1). Furthermore, P8 thinks that if too many rules are enabled in a project, people will not trust the configurations: "*They will think that it is a bureaucracy and that it's not important.*" (P8). In any case, both P1 and P8 prefer to only enable rules that can prevent errors. P14 further explained that it is simply easier to maintain a smaller configuration file compared to a larger one.

#### 4.3.2.6 Effort of Enforcing a Rule

Participant P15 described that he commonly enables a set of rules, *e.g.*, a known preset, and then sees how it works out for the project. If some of the rules are starting to be bothersome for the project, *e.g.*, needing to be disabled with inline comments or if too much refactoring is required to fit the rule, it is permanently disabled: "*Just start to use it and see how much pain it causes, where it's beneficial. But usually it's turning things off when it's apparent that it's creating more effort than it really helps.*" (P15). He describes the process as a feedback cycle where it is important that contributors agree on the rules that are used: "*The disagreement between people is very important, you have to get everyone on the same page.*" (P15).

For rules that relate to possible errors, P4 discussed a similar approach where he considers whether a particular rule will be useful for the project or if they will need to disable it in multiple locations in the code. If it needs to be disabled frequently, it is not worth it to enforce it.

#### 4.3.2.7 Most Voted Style

Participant P7 reported that in a new project he would most likely go with the code style that is the most common one amongst the developers in the team, consequently adding rules that enforce that style. Generally when working with a new team, the first discussion he often has with them is regarding which code style people are used to.

#### 4.3.2.8 Consistent Rules

Lastly, some developers do not care all that much about which rules are actually enabled (P2, P6, P10): "*I almost don't even care what the rules are, I have some opinions, but I'm much more interested in there just being consistent rules, than having a point of view about any particular rule.*" (P10). There just has to be some fixed set of rules to enforce consistency and to prevent unnecessary time being spent on discussions: "*Having a linter forces us to make choices, even if it's arbitrary choices in some situations.*" (P2).

**RQ$_2$ How are rules selected for a project?**
The methods that developers use to configure ESLint are to use an existing preset, choose rules that fit the project's style, have the linter automatically generate a set of rules, enable rules that come up in discussions during code review, have minimalistic configurations, choose rules that involve the least effort to follow and to go with the most common style within the development team.

### 4.3.3 RQ$_3$ Challenges of Using a Linter

While the interviewees were generally happy with ESLint and its capabilities, they described some challenges with using the tool which are discussed here.

#### 4.3.3.1 Create and Maintain Configurations

Several participants mentioned that it was challenging to create the initial configuration or to keep it updated (P1, P3, P5, P8, P12). Only two participants reported that they had examined all available rules when they originally created the configuration file (P3, P9). This involves evaluating a set of 236 rules that ESLint has available which can be a tedious process: "*Most of it was read through every rule, it was kind of a very painful process to set it up.*" (P3). Another participant used very similar wording when it came to setting up the tool: "*Sometimes a pain to set up in your editor with the right configuration.*" (P5). Meanwhile, the other participant that manually created the configuration file claimed that it was actually quite easy to set up (P9).

Others have been frustrated with keeping their configurations up to date after they have been created, especially when using presets (P1, P8, P12). When the presets are updated, there are often new rules that are enabled or older rules that are changed which can cause a high volume of new warnings or errors (P12). P12 explained that these changes are sometimes beneficial and he happily fixes the warnings, but at other times the changes are not useful and the rule settings have to be overwritten. Moreover, P1 discussed that it can be frustrating when the presets update very frequently and the code has to be changed often because of it. However he also likes that the presets keep him updated of new rules regarding new JavaScript features.

#### 4.3.3.2 Enable Rules in Existing Projects

Six participants (P1, P3, P4, P7, P10, P13) mentioned that it can be difficult to start using a linter or to enable new rules in an existing codebase. If the rules that are enabled cause many warnings or errors to occur, substantial effort is needed to go over all existing code to fix every reported instance. There can even be such a large volume of existing code that it would simply take too much time to review: "*The problem with [the project] is that it's really old and big, so for that we don't have the luxury of turning on whatever rules we want to because we're never going to be able to update a lot of our code to support them.*" (P1).

Even if it is possible to fix all warnings, it can be risky to change old code to conform to these rules since it is easy to introduce new bugs in the meantime (P3, P7, P10). Knowing the original intent of the code can be very difficult and can take considerable time to try to understand (P7). P7 discussed an example where there were many instances of two equal marks being used in the code instead of three, which resulted in many errors from the linter. In these cases it could have been intentionally written so or by accident, which would need to be carefully verified and tested. The risk and effort of going back and changing the old code might therefore not be worth it: "*Cleaning up for just clean up, just to enable it, I think is risky. I've been bitten by that a couple of times.*" (P3). Enabling a linter in a project to begin with can be a lot easier: "*A linter is great if you start with that and you enforce all those rules and the idea is that you will never run into ambiguous code.*" (P7). However with older projects it can be more difficult and even dangerous (P7).

It can also be challenging to choose which rules to go by for an existing project. A common method is to use a preset and then modify it slightly based on the project style. In a large existing codebase the code style can however be very evolved and too different to the available known presets such as the popular Airbnb (P10). It is a different case with new projects where the developers can choose one style from the beginning and always follow it.

In large existing projects with many collaborators it can also cause conflicts and frustration when new rules are enabled (P4, P13). P4 was working at a company with 60-70 developers, all working on the same code every day. Usually there are multiple pull requests open at the same time and there will be many merge conflicts when a new rule is suddenly enabled, since it will likely affect code everywhere in the project. Furthermore, in P13's project it caused frustration with people to enable many rules all at once since the developers were not accustomed to them before. Instead, it was decided to do it slowly by enabling only one rule at a time to give developers a chance to get used to the new rules.

#### 4.3.3.3 Agree on Rules in an Industrial Setting

In an open source project it is relatively easy to build consensus around what people want to do (P4, P15). If someone wants a new rule to be enabled, then only a few main contributors need to say yes and it is decided upon. Other maintainers usually follow what the project leaders propose (P4, P15). In a business environment it can be a very different case where everyone in the team has a say in the matter: "*There are 60 people with their own opinion about how code should be written.*" (P4). P4 further explains that it can be especially difficult to introduce rules on stylistic issues since it is not considered to be important and it can be hard to justify why you would inconvenience people to adhere to new rules: "*It may create more tension than it does actually solve problems*." (P4).

However, the power of the lead developers in open source projects can also be a negative factor. If a contributor wants to enable a new rule, it often only needs one reviewer to say no, so that it will not be accepted (P2). In that case the pull request might get rejected or it has to go through the core technical team for a discussion which might not be worth the trouble, only to enable a new rule (P2).

#### 4.3.3.4 Enforce Developers to Follow the Rules

When a rule is configured in ESLint it can be enabled as an error, disabled or set as a warning. As explained before, having a rule enabled as an error will break the build when ESLint is a part of the build process. The majority of the participants used only or mostly errors and rarely warnings. These settings are used by the participants for different purposes, such as indicating the criticality of a rule (P15) or by using warnings as an adaptation period when enabling new rules (P4, P7, P13). Other participants liked to use warnings in the development process so that their build would not be interrupted when working on unfinished features (P9, P12, P15). The possibility of having warnings is a nice feature that allows for enforcing rules but without breaking the build. There is however a problem with using warnings: multiple participants claimed to use only errors since warnings would simply be ignored by developers (P2, P3, P4, P13). In addition, according to these participants, when warnings live for a long time in the codebase, people will start to devalue them and leave them behind. Developers do not feel any responsibility for the warnings and might think "*there were some warnings when I checked it out, not my job to fix it, I'm going to check it back in with warnings.*" (P4). P4 further explains that especially in the presence of many warnings, developers will not even read them and simply leave them behind. To enforce removal of the warnings, they therefore have to be set as errors that actually cause a build to fail.

#### 4.3.3.5 Dynamic Features of JavaScript

JavaScript has been described as harsh terrain for static analysis because of its dynamic nature [138]. Static analysis for JavaScript has thus been criticized and said to be limited since it can not account for runtime behavior [94]. The majority of the participants would indeed like ESLint to be able to do more, but they however think that the current version is very acceptable since they choose themselves to work with a dynamic language (P2, P5, P6, P7, P8, P9, P10, P11, P15). When P9 was asked whether he misses dynamic analysis from the linter he replied: "*Of course, but I don't particularly think that is ESLint's fault, so much as a language defect. Due to the dynamic nature of JavaScript, static analysis ranges from extremely difficult to impossible. I don't expect ESLint to be able to fix that.*" (P9). Regarding type checking, there are trade-offs in choosing JavaScript as a programming language and there is a reason why some languages are not strictly typed (P2, P15). If static typing is something that a developer wants, he or she should rather use TypeScript or some statically typed language (P2, P10, P15): "*Each developer has to make a decision on if they want to work in a typed language or in an untyped language and the trade-offs of that kind of lead you to the path of what tools can provide.*" (P15). The majority is therefore quite satisfied with what the linter can do and do not expect it to be able to detect more of the dynamic parts of the language.

Other participants were more bothered by the fact that a linter can not analyze the dynamic parts of the language, where the problem was mostly centered around JavaScript's weak typing (P3, P4, P13, P14). P3 reports that he has spent substantial time on testing various mix-ups with strings and numbers, for which he would either like ESLint to warn about types that change or would like to switch from using regular JavaScript to TypeScript.

#### 4.3.3.6 The Presence of False Positives

A common problem with static analysis tools is the high volume of false positives [164, 108, 71, 100]. Studying the usage of ASATs, Johnson *et al.* [108] found that the presence of false positives is indeed one of the largest barriers in using such tools. Opposed to previous literature, the majority of the interviewees reported that they do not in general experience false positives while using ESLint (P5, P6, P9, P10, P11, P12, P14, P15). Three participants mentioned that they had experienced some false positives in the past which have now all been fixed (P5, P12, P14). Moreover, P14 only experienced a false positive once where he then reported the issue which was fixed in a matter of a few hours. Two participants claimed to experience somewhat frequent false positives (P10, P13). However, in both cases they blame it on other elements than ESLint itself, such as their code editor. In fact, P13 migrated from JSLint and JSHint to ESLint because he perceived much fewer false positives with ESLint.

Furthermore, the term false positive can be understood in two different ways, either a wrongly reported warning or a true warning that is not considered by a developer to improve the software under analysis [67]. Some participants discussed this type of false positive, which is when the linter flags a certain instance but the developer thought it was appropriate to break the rule in that particular case (P3, P6, P7, P9, P11). Participants P9 and P11 however considered it to be positive to receive a warning in these cases. P9 explained that it makes him think about whether breaking the rule is actually a good idea or not: "*That is OK, the linter is there to make sure we are being thoughtful about our choices.*" (P9). Furthermore, the presence of comments that disable rules can be useful for other developers to know the intention of the code: "*That's helpful because now other people reading the code will know this is definitely intentional. So I think it's useful for the reader and ESLint is great that it's flexible like that.*" (P11). These comments can therefore serve as documentation of some sort to help others understand the intention of the code.

**RQ$_3$ What are the challenges in using a linter?**
The main challenges that developers face are to create and maintain configurations, enable rules in an existing codebase, agree on which rules to use in an industrial setting and to enforce developers to follow rules without breaking the build. They are generally not concerned with the lack of dynamic analysis and generally do not experience false positives.

## 4.4 Discussion

### 4.4.1 Reasons for Using a Linter

The interviewees provided multiple reasons as to why they use a linter in their JavaScript projects. The most commonly mentioned reason was to maintain code consistency, where every single participant claimed to use the tool for that purpose. However, when asked about the importance of this aspect, it was rated low compared to other types of rules than these stylistic ones. The reason that was mentioned by the majority of the participants to be the most critical one was to prevent errors, and especially errors that have to do with the dynamic typing of JavaScript. So even though it was thought to be important to enforce a consistent code style, it is not highly prioritized as catching possible bugs in the code is far more important. In general, the interviewees discussed the importance of four ESLint categories: Possible Errors, Best Practices, Variables and Stylistic Issues, appearing to be the main ones of interest.

### 4.4.2 Configuring Linters

Developers use various methods to choose the rules to include in configurations. Commonly used methods are to use a preset and to choose rules that fit the already existing style of a project. In this section we saw the methods that developers use to choose rules but not which types of rules they actually enable. Meanwhile, in the previous section we saw the reasons as to why developers use a linter and the types of rules they find to be important. During the interviews it was evident that these two topics are very much connected in the way that their motivation in using a linter is what, amongst other things, guides them in choosing the rules that they include in their configurations. If a developer's main reason for using a linter is to prevent errors, he or she will try to enable rules that can catch critical errors. On the other hand, if a developer uses a linter to maintain consistent code, he or she will try to enable rules that enforce a specific code style. Throughout these discussions with the participants they gave examples of rules that they consider to be important, as well as being explicitly asked to mention a few rules that they find important to include in configurations. The rules that were most commonly mentioned by the interviewees are displayed in Table 4.3, where the top ones have already been discussed in the above analysis. The top five most commonly mentioned rules all come from the Variables and Possible Errors categories while the rest also includes rules from Best Practices and Stylistic Issues. Furthermore, Table 4.4 shows the number of unique rules that were mentioned for each of the seven categories, where Possible Errors was the most commonly referenced one.

| **Rule** | **Category** | **Frequency** |
|---|---|---:|
| no-unused-vars | Variables | 9 |
| no-undef | Variables | 5 |
| no-unreachable | Possible Errors | 5 |
| no-console | Possible Errors | 4 |
| no-dupe-keys | Possible Errors | 4 |
| eqeqeq | Best Practices | 3 |
| max-len | Stylistic Issues | 3 |
| semi | Stylistic Issues | 3 |
| coma-dangle | Stylistic Issues | 2 |
| indent | Stylistic Issues | 2 |
| no-dupe-args | Possible Errors | 2 |
| no-fallthrough | Best Practices | 2 |

Table 4.3: Most important rules according to interview participants

| **Category** | **Total mentions** | **Unique rules** |
|---|---:|---:|
| Possible Errors | 19 | 8 |
| Variables | 15 | 3 |
| Stylistic Issues | 14 | 7 |
| Best Practices | 10 | 7 |
| Strict | 1 | 1 |
| ECMAScript 6 | 1 | 1 |
| Node.js & CommonJS | 1 | 1 |

Table 4.4: Important rules mentioned per category

### 4.4.3 Challenges of Using a Linter

Despite the positive attitude towards ESLint, there are some challenges that developers face. The most commonly mentioned challenges were to enable rules in an existing codebase and to create and maintain configurations for a project.

Other challenges that have been studied in the literature, namely the lack of dynamic analysis for JavaScript code and the presence of false positives, do not seem to be very problematic for these developers. For dynamic analysis, they are aware that several features are missing from the linter but generally do not seem to be bothered by it. For false positives, they hardly ever experience them. This difference to previous literature might result from the relatively simple type of analysis that is performed by ESLint and other JavaScript linters. By using more complex analysis methods to identify more intricate issues, the risk of reporting false positive increases, and additionally the risk of users not understanding the issues and mistaking true positives for false positives [71]. The issues detected by ESLint are of a more simplistic nature where warnings are typically reported on only a single line of code, *e.g.*, a variable that is not initialized. This is, for instance, very different from code smell detection tools for Java; as many code smell definitions are somewhat abstract, tools rely on heuristics [127] which can lead to false positives. For example, to detect a God Class, PMD's heuristic relies on the combination of several metrics, namely *Weighted Method Count*, *Tight Class Cohesion* and *Access to Foreign Data* metrics [122, 47].

## 4.5 Conclusion

We interviewed 15 developers about their perceptions of using a linter for JavaScript code. These developers worked on notable OSS projects on GitHub and were considered to be experts on using JavaScript linters. We used a qualitative research method inspired by Grounded Theory where we allowed new ideas to emerge from the data. We explored and discussed why and how developers use linters along with what challenges they face. While most developers use a linter to maintain code consistency, it is more important to use it to prevent possible errors and typing mistakes in code. Using a linter is also beneficial to save time and effort on manually reviewing code submissions for software projects. Developers employ various ways to decide on which rules to use in configurations for a linter, including using a preset and enabling rules that fit the current style of a project. While it is generally not difficult to use ESLint, it can be challenging to maintain configurations and to enable new rules in existing projects.

# Chapter 5

# Exploring Linter Configurations in JavaScript Projects

In the previous part we examined in a qualitative way why and how developers use linters. For this part of the study, a quantitative analysis is performed to know exactly how developers configure their linters and what the most common configuration patterns are. For this purpose, we analyzed 9,548 ESLint configuration files from JavaScript projects on GitHub. To find these projects, a number of 86,366 JavaScript projects were examined in total. We analyze how much configurations are applied by developers and whether they rely more on pre-made settings (presets) or their own configurations, which helps ASAT makers to understand how their tools are used. We further see which types of rules are most commonly used for these projects, which assists developers in choosing which rules to include in their projects, along with helping tool makers to see which which type of code analysis to focus on.

## 5.1 Research Questions

The overall question that drives this chapter is *how do developers configure linters?*. For that purpose we study the prevalence of configurations and see which types of rules are most commonly used by developers. We look at the data from two different angles: projects that use a preset and projects that do not use a preset. This distinction is made so that we can examine the prevalence of rules without the unknown effects of the settings that the presets entail. Furthermore, when observing projects that do in fact use a preset, we obtain a different type of insight where it is possible to see what functionality developers think is important to add to the presets or even to remove from them. The rules that are examined in this analysis are only the native rules offered by ESLint, and not custom made rules or rules that belong to external plugins. More specifically we examine:

**RQ$_4$ How much do developers configure ESLint?**
We examine how much configurations are applied in terms of using pre-made settings (presets) or specifying rules that are chosen specifically for a project. We can

then evaluate how much developers want and need to maintain their own custom configurations, or whether they prefer to use settings that are recommended by external parties. For this aspect we focus on the following more detailed questions.

**RQ$_{4A}$ How are presets and plugins used?**
We analyze the prevalence of using presets and plugins and see which of them are the most popular ones. More emphasis is put on presets rather than plugins, since the usage of presets can tell more about how the general set of rules is prioritized. The usage of plugins however explains more about which JavaScript libraries are used and require linting.

**RQ$_{4B}$ How frequently are rules specified?**
We examine the prevalence of rules being specifically specified in configurations and differentiate between rules that are enabled, disabled or set as warnings. We further see how the prevalence of specified rules is different when presets are used or not.

**RQ$_5$ What are the most common configurations for ESLint?**
We see which types of rules are most commonly enabled, disabled or set as warnings in configurations, differentiating between projects that use presets and those that do not. By doing so, we can see which aspects of linting are the most important to developers, *e.g.*, whether it is regarding catching bugs or maintaining a specific code style. For this we ask the following questions.

**RQ$_{5A}$ Which are the most commonly used categories from ESLint?**
We analyze the prevalence of each category from ESLint and see which are the most commonly used ones.

**RQ$_{5B}$ Which are the most commonly used rules from ESLint?**
We see which are the most commonly used rules, both overall and within each individual category.

## 5.2 Methodology

A tool was created that receives a list of projects to examine where ESLint configuration files are first collected and then parsed and analyzed. This tool is called ESLint-config-analyzer and is available online via GitHub [153]. In the following we describe the implementation and limitations of these two parts of the tool but first we describe the dataset and how it was retrieved.

### 5.2.1 Dataset

To collect projects we chose GitHub as a data source due to the high number of available JavaScript projects and the ease of retrieving the data [110]. The next section describes how the final dataset was chosen and retrieved, followed by a description of the main characteristics of the dataset, such as the sizes of the projects and initial results on the prevalence of linter usage.

#### 5.2.1.1 Data Collection

The original data selection consists of all JavaScript projects on GitHub that have at least 10 stars and are not forks[1] of other projects. The purpose of starring a project is to keep track of projects that a user finds interesting or simply to show appreciation to a repository [90]. By only including projects with at least 10 stars the intention is to only analyze those repositories that can possibly count as "real" software projects. Kalliamvakou *et al.* [110] showed that a large portion of GitHub repositories are not for software development but for other functions such as for experimental or storage purposes. It is expected that repositories that were created in some kind of experimentation or testing purposes only, or pet projects that were started and abandoned, will not receive 10 stars from other users. Furthermore, forks of other projects were excluded to avoid having duplicate configuration files in the dataset. This resulted in a number of 86,366 projects being collected to analyze.

To retrieve data on these projects, we used Google BigQuery [4] on the most recent GHTorrent [96, 31] dataset on GitHub projects from April 1st, 2017. The precise SQL query that was used to obtain the data can be found in Appendix B.

After retrieving the 86,366 project entries, some additional filtering was performed on the dataset. First of all, there were examples of duplicate entries in the dataset where they point to the same GitHub API URL for the project. This can happen when the project name has been modified or when the owner has been changed for a repository, in which case a new entry is created in the GHTorrent dataset. Before analyzing the projects, they were all collected together in one set where duplicates were removed (entries having the same API URL). The duplicate entry that had a more recent date for its last commit was selected to be left behind. This filtering resulted in 1,596 projects being removed from the dataset. Secondly, even though the query includes a statement to not include deleted projects, some repositories could not be accessed. In seven cases, an HTTP error status code of 451 was returned when trying to access the repository. That means that the project is unavailable for legal reasons, more specifically due to possible copyright violations. More commonly, or in 871 cases, the repository's URL could not be found, returning an HTTP status code of 404. Most likely these are projects that have been deleted since the dataset was uploaded on Google BigQuery, but almost eight weeks had passed after the data was uploaded until the query was executed. Due to this filtering, a number of 878 additional projects could not be analyzed, resulting in a final number of 83,892 possible projects.

#### 5.2.1.2 Data Characteristics

Besides filtering out forks, duplicates and deleted projects, no additional filters were applied such as regarding project size or activity. We decided not to filter the projects by a minimum size, as the intention was to analyze all different types of JavaScript projects; big or small; collaborative or personal. However it could therefore be the case that the resulting dataset includes projects that are not suited to ever use a linter, *e.g.*, a repository that is not a software but simply a collection of scripts or tips for developers. Additionally, these could be projects

[1] A "fork" is a personal copy of someone else's project on GitHub.

that have not been active for several years, and perhaps even not active since ESLint was created in June 2013.

Consequently, to know more about the dataset, the date of the last commit was collected from GHTorrent and the GitHub API was used to retrieve the size of every project. The sizes of the projects are expressed in kilobytes (KB) where the first, second and third quartiles are the following: 126 KB ; 364 KB ; 1,928 KB. Moreover, 55,9% of the projects are 500 KB or smaller. For the last commit date of the projects, 7.4% of the projects had their last commit before ESLint was first released (v0.0.2) in June 2013 and 35.4% of the projects before the first major release (v1.0.0) in July 2015[2]. Thus when analyzing the prevalence of linter usage in JavaScript projects or even more specifically, ESLint usage, some projects in the dataset are less likely to have used a linter. There are projects that might not be typical candidates to use a linter such as programming guides (example at [54], size 618 KB) or code for tutorials (example at [33], size 129 KB). Other projects have not been active since ESLint was created and have thus not had an opportunity to use the tool. Nevertheless, these projects were not filtered out as we wanted to avoid making assumptions about the dataset and for it to be as broad and general as possible.

As shown in the background chapter, the most popular linter according to the number of npm installs is ESLint, followed by JSHint. While collecting the configuration files for the projects that use ESLint, information was gathered about whether these projects use other linters, and if so, which. How this information was retrieved is explained in the next section. Table 5.1 shows an estimation of the usage of four linters, namely, ESLint, JSHint, JSCS and Standard, where ESLint was the most commonly used linter followed by JSHint. Moreover, in total 24.5% of the analyzed projects used a linter. The tool also analyzed whether projects use more then one linter, with the results shown in Table 5.2. Only 9.1% of the projects that use a linter, use two or more linters, where 4.3% use ESLint and another linter. The total percentage of projects using any of these linters seems to be higher for those that have more stars. Table 5.3 shows the number of projects that use any of these linters while observing only the top starred projects, such as the top 10 or top 1,000 projects. The percentage of projects using a linter steadily decreases when observing more projects, going from 70.0% for the top 10 projects to 28.5% for the top 30,000 projects.

| **Linter** | **Projects with linter** | **% of all projects** |
|---|---:|---:|
| ESLint | 9,548 | 11.4% |
| JSHint | 9,447 | 11.3% |
| Standard | 1,651 | 2.0% |
| JSCS | 1,578 | 1.9% |
| **Total** | **20,292** | **24.2%** |

Table 5.1: Estimation of the number of projects using ESLint, JSHint, Standard and JSCS

[2]The exact dates that were used for this comparison were 08:00 June 30th 2013 [17] and 08:00 July 15th 2015 [16].

| Linters used | Number of projects | % of all projects |
|---|---:|---:|
| 1 | 18,450 | 22.0% |
| 2 | 1,753 | 2.1% |
| 3 | 88 | 0.1% |
| 4 | 1 | 0.0% |
| **Total** | **20,292** | **24.2%** |

Table 5.2: Estimation of the number of projects using multiple linters

| Top projects | Projects with linter | % of top projects |
|---|---:|---:|
| 10 | 7 | 70.0% |
| 100 | 65 | 65.0% |
| 300 | 185 | 61.7% |
| 1,000 | 535 | 53.5% |
| 3,000 | 1,371 | 45.7% |
| 5,000 | 2,082 | 41.6% |
| 10,000 | 3,675 | 36.8% |
| 20,000 | 6,306 | 31.5% |
| 30,000 | 8,554 | 28.5% |
| **Total** | **20,292** | **24.2%** |

Table 5.3: Estimation of the number of projects using a linter

### 5.2.2 Collecting Configuration Files

In order to analyze the configurations of projects that use ESLint, one first needs to collect all the configuration files. The following sections describe how the tool accomplishes this goal and what its limitations are.

#### 5.2.2.1 Implementation

Querying for all projects with at least 10 stars on GitHub resulted in a list of 86,366 JSON objects with information about each project, including the project's name and GitHub API URL. The first step in retrieving the configuration files is trying to access the project's API URL. If an error is returned, either the access to the project is restricted or more commonly, the project does not exist anymore. Additionally before moving forward, duplicate entries are ignored as explained in the dataset section. If it is possible to access the project, the tool searches for indications that the tool uses ESLint, JSHint, JSCS or Standard by identifying configuration files for each linter. If a configuration file is found for ESLint then it is retrieved, while for the other linters the tool merely notes down their presence. As a project can take advantage of multiple linters, the tool searches for the presence all linters in every project.

More specifically, for each linter the tool searches for a configuration file with a specific known name and file ending. As an example, the tool searches for ESLint configurations files with the name `.eslintrc` and one of the following file endings: `.js, .yaml, .yml, .json`. Additionally, the ESLint configurations can be present in a project's `package.json` file, leading to all package files being examined as well in search for a key that contains the configurations. This process is done in a similar fashion for both JSHint and JSCS. On the other hand, as the Standard linter's philosophy is to not be bothered with any configurations, another method had to be used to identify the linter's usage. In this case the `package.json` is parsed to search for a dependency to the Standard npm package. Finally, the GitHub API is used to retrieve the size of every project. The total running time of the tool to collect the configuration files for the 86,366 projects was 29 hours and 36 minutes.

#### 5.2.2.2 Limitations

The main limitation of this part of the tool is that it can possibly miss out on some ESLint configuration files. The configuration file is typically located in the main directory of a project (as it will then be used for the whole project), so in order to save execution time and to simplify the tool, it is the only location where the tool searches for the file. It could however be the case with some projects that they place their configuration file in a sub directory of the project and pass it to the linter with the command line. This limitation does however not have any significant effects on the results, since the only real implication is that the dataset for analyzing the configuration files is smaller than it could have been. A trivial limitation is that the numbers are skewed about the prevalence of using ESLint, as reported in the previous section about the dataset's characteristics. That information is however not of main interest for this study.

Continuing on the topic of the prevalence of linter usage, there are limitations in acknowledging the presence of other linters as well. As the tool does not check for all linters that exist, no assumptions can be made about the overall prevalence of linter usage. For example, the tool does not account for usage of JSLint, but it was decided to disregard that linter since the usage is less frequent than of the other linters and it was more difficult to identify its usage. Moreover, for some linters it is not necessary to have a configuration file in place and it is also possible to specify a configuration file with other names than the default recognized formats. For these reasons the usage could as well be understated. On the other hand, the usage in some cases can be overstated. Even though a project has a configuration file for a specific linter, it does not guarantee that the linter is actually used. As an example, the tool found that one project has four linters enabled, having configuration files for ESLint, JSHint and JSCS along with having Standard as a dependency [41]. On closer inspection one can see that only one of them seems to be currently in use (and even additionally uses TSLint that is intended for TypeScript)[3]. Further measures could have been taken to make this analysis more accurate, such as analyzing the dependencies of all projects along with all its pre-running scripts. That would however still not guarantee true results and was considered out of scope, as this is not the main purpose of the study. For

[3]When manually inspecting the history of `package.json` for the project [42], one can see that the dependencies for ESLint, JSHint and JSCS were removed at one point and also removed from all pre-running scripts. Instead, solely Standard was enabled.

these reasons, the prevalence numbers are presented as an estimation of the usage and not as definite findings.

To increase confidence in the current implementation to identify the usage of ESLint, JSHint, JSCS and Standard, unit testing was applied. Actual use cases of all covered situations to detect these linters were used as input to see if the tool identified them correctly.

### 5.2.3 Parsing Configuration Files

In this part of the tool the configuration files are parsed to retrieve all information that is of interest. In the following sections the implementation of this will be described along with its possible limitations.

#### 5.2.3.1 Implementation

From the previous part of the tool, a list of URLs is received as input, each pointing to an ESLint configuration file of a JavaScript project. These configuration files can be in several forms, namely a `package.json` file or an `.eslintrc` file in any of the formats that are allowed. Each of these formats are parsed differently to correctly obtain the information of interest. The tool searches for the presence of the properties that address the presets, plugins and rules that are used. Special care is taken to not mistake lines such as comments that include these keywords for lines that are actually of interest.

To differentiate between the different file formats (*e.g.*, `.js` or `.yml`), it is not enough to simply check the file ending of the configuration file because of the old convention of not having any file ending as explained before in the background chapter. Many of the projects still use this deprecated format so it was necessary to make a distinction between all configuration files during the parsing of the files where the patterns of each format is identified.

To parse the rules that are specified, the tool needs to apply some filtering when identifying a rule and its settings. There is a fixed amount of rules available from ESLint and only those are studied. Custom rules and rules from external plugins therefore need to be recognized and excluded. Moreover, a rule has its name as a property and different kinds of settings as its value, as described in Chapter 2. Enabling a rule can be done with setting a value `0` to the rule name but also with a value `error`. Then there are multiple other settings that can follow which are different for each rule (*e.g.*, setting which kind of indentation to enforce). The tool only extracts the value that enables, disables or sets the rule as a warning, and ignores the rest. Presets and plugins are however mainly parsed without any filtering as their names are unknown beforehand.

Eventually, two configuration files could not be retrieved as a status code of 503 was returned when trying to read the files, meaning that the service was temporarily unavailable. The total running time of analyzing the 9,548 configuration files was 21 minutes and 9 seconds. This second part of the tool is therefore much faster in execution as it does not need to make any network calls like in the previous part.

#### 5.2.3.2 Limitations

The parser implementation has some limitations as it may not parse all code writing styles correctly. The requirements for the tool were set by reading the documentation from ESLint [84] and examining numerous different configuration files. This was done to try to discover all possible methods in writing the configurations along with all edge cases that the parser would need to handle. It is however likely that some specific ways of writing were not discovered in this process, resulting in the parser perhaps missing out on some elements. Moreover, even though the parser does handle different types of formatting, it is difficult to foresee and analyze all formats that can possibly be written by developers. A known limitation of the parser is that is does not correctly parse code that is not well formatted, *e.g.*, when no line breaks are made between properties in a file. In these case, the parser does not recognize the settings as it can not correctly identify the properties of interest, resulting in fewer rules or presets being reported. Another situation which the parser can not handle is when code is used as a value for a key in the configurations such as using a ternary operator to decide on a value. In these (rare) cases, the parser can not differentiate between the name of an actual preset or a piece of code, which results in the wrong value being recorded.

To mitigate possible limitations regarding misidentifications of presets, plugins and rules, the output of the tool was thoroughly examined to detect any discrepancies such as presets or rules that do not exist. Nearly all cases of these were fixed in the parser but a few wrong values were left unfixed as described above. Additionally, various known configuration formats are unit tested. Furthermore, several random samples of actual configuration files were manually tested to see whether all presets, plugins and rules were correctly identified.

## 5.3 Results

We present the results of mining all configurations files, in the order as the research questions were previously presented.

### 5.3.1 RQ$_4$ Prevalence of Configurations

As stated before, no default configurations come with ESLint. It therefore has to be configured in some way, most commonly by specifying a configuration file. This configuration can include presets, plugins and/or some rules that are enabled, disabled or set as warnings. We first discuss the prevalence of presets and plugins, followed by an analysis on the prevalence of individual rules that are specified.

#### 5.3.1.1 Presets and Plugins

Out of the 9,548 projects that were analyzed, 6,413 or 67.2% used a preset in their configurations, with a total of 6,967 presets being used (some projects using more than one preset). The 10 most popular presets are displayed in Table 5.4. It is evident that only a handful of presets are extremely popular amongst these projects. Only the five most popular presets account for 67.2% of all presets that were used. The recommended settings from ESLint [84] is the most popular preset, followed by the Airbnb preset [13]. It is interesting that in the top five places, there are two presets by Airbnb: the regular one (`airbnb`) and the base one (`airbnb-base` [12]). These two presets both use the import plugin (`eslint-plugin-import` [20]) but the difference is that the regular `airbnb` preset also uses two plugins for React (`eslint-plugin-react` [24] and `eslint-plugin-jsx-a11y` [22]). If these two presets are counted together, they have a total of 1,723 instances, making `airbnb` and `eslint:recommended` (2,226) by far the most popular ones (56.7% of all presets that are used).

| **Preset** | **Projects** | **% of all presets** |
|---|---:|---:|
| eslint:recommended | 2,226 | 32.0% |
| airbnb | 1,166 | 16.7% |
| standard | 627 | 9.0% |
| airbnb-base[4] | 557 | 8.0% |
| google | 104 | 1.5% |
| standard-react | 81 | 1.2% |
| prettier | 58 | 0.8% |
| rackt | 49 | 0.7% |
| react-app | 48 | 0.7% |
| semistandard | 44 | 0.6% |
| **Total** | **4,960** | **71.2%** |

Table 5.4: The 10 most popular presets, showing the number of projects using each preset and the percentage of the times the preset is used out of all used presets

The usage of plugins is not as common as for presets, where 2,611 projects included a plugin in their configurations, or 27.3% of all projects that use ESLint. The total number of plugins that were used is 3,556, with the 10 most popular plugins being shown in Table 5.5. It is evident that `eslint-plugin-react` [24] is by far the most popular plugin, accounting for 47.0% of all plugins that are used. This plugin includes rules that are specific for the React library which has become very popular since its release in July 2013 [50], being the 4th most starred JavaScript project on GitHub with more than 68,000 stars and having over 44 million npm downloads [160] [5]. It is possible to use plugins in two different ways, either by specifying each individual rule that should be included from the plugin, or by using it in whole as a preset. A number of 440 projects used a plugin as a preset, and using in total 561 plugins for this purpose. Table 5.6 displays the plugins that are most commonly used as presets. Again, the `react` plugin proves to be the most popular one.

[4]There were two entries for the Airbnb base configurations, namely `airbnb-base` and `airbnb/base`, with 378 and 179 instances, respectively. The latter is however only a deprecated npm entry point for the previous, so these two were counted as one preset in the analysis.

[5]Information as of June 5th 2017.

| Plugin | Projects | % of all plugins |
|---|---:|---:|
| react | 1,671 | 47.0% |
| import | 298 | 8.4% |
| babel | 294 | 8.3% |
| html | 174 | 4.9% |
| flowtype | 166 | 4.7% |
| standard | 94 | 2.6% |
| mocha | 91 | 2.6% |
| jsx-a11y | 87 | 2.5% |
| promise | 87 | 2.5% |
| prettier | 64 | 1.8% |
| **Total** | **3,026** | **85.1%** |

Table 5.5: The 10 most popular plugins, showing the number of projects using each plugin and the percentage of the times the plugin is used out of all used plugins

| Plugin | Projects | % of all plugins |
|---|---:|---:|
| react/recommended | 181 | 32.3% |
| import/errors | 88 | 15.7% |
| import/warnings | 56 | 10.0% |
| flowtype/recommended | 47 | 8.4% |
| ember-suave/recommended | 45 | 8.0% |
| node/recommended | 24 | 4.3% |
| meteor/recommended | 15 | 2.7% |
| ava/recommended | 14 | 2.5% |
| import/recommended | 9 | 1.6% |
| jasmine-jquery/recommended | 4 | 0.7% |
| **Total** | **484** | **86.1%** |

Table 5.6: The 10 most popular plugins that are used as presets, showing the number of projects using each plugin as a preset and the percentage of the times the plugin is used out of all used plugins as presets

**RQ$_{4A}$ How are presets and plugins used?**
Presets are used by 67% of all ESLint projects where the most popular ones are the recommended settings from ESLint and a style guide maintained by Airbnb, which account for over half of all presets that are used. Plugins are used by 27% of all projects where a plugin for React is by far the most popular one.

#### 5.3.1.2 Frequency of Specified Rules

Here we examine the frequency of rules that are specified in configurations, for each type of rule setting and differentiating between projects that use presets and those that do not. The quantity of rules that are configured are displayed in Table 5.7. More specifically, Table 5.7a shows the quantity of enabled rules, Table 5.7b the quantity of rules that are set as warnings, Table 5.7c the quantity of rules that are disabled and lastly Table 5.7d shows together all rules that are specified. Each table is divided in three groups: 1) projects that use a preset, 2) projects that do not use a preset and 3) all projects.

First of all, it is evident that warnings are in general used the least, where there are three warnings used on average per project. There are more than twice as many rules that are turned off, or on average seven per project. Rules are however most frequently turned on, or on average 17 times per project. It is also visible that projects that do not use presets, specify more rules in general than projects that use a preset. More specifically, 95% of the projects that do not use a preset, use at least one rule and specify 58 rules on average in their configurations (the 5% that do not have any rules are reviewed in the discussion). Furthermore, 70% of the projects that do use a preset have at least one rule specified, and use 10 rules on average. For these projects, the same percentage of projects enable and disable at least one rule (53%) but more rules are used on average (seven rules enabled and three rules disabled).

**RQ$_{4B}$ How frequently are rules specified?**
Projects that do not use presets specify a high number of 58 rules on average. Those that do use presets specify 10 rules on average where 70% of these projects include at least one rule. For all projects it is most common to enable rules as errors and rare to set rules as warnings.

### 5.3.2 RQ$_5$ Common Configurations

ESLint provides a basic rule set that is grouped into seven different categories, as explained in the background chapter. We examine which categories are most commonly enabled or disabled, divided between projects that use presets and those that do not. Similarly we examine which individual rules within each category are most commonly used.

#### 5.3.2.1 Common Categories

Tables 5.8 and 5.9 show the average number of rules that are enabled and disabled, respectively, from each category per project along with how many projects use at least one rule from each category.

**Enabled Categories**. For projects that do not use presets, four categories are enabled in more than 50% of all projects: Possible Errors, Best Practices, Variables and Stylistic Issues (Table 5.8). The category Stylistic Issues is by far the most commonly enabled one, where 82% of all projects that do not use a preset, enable at least one rule. The next category in line is Best Practices where 62% of projects without a preset enabled at least one

| | **Total rules** | **Average** | **# projects** | **% projects** |
|---|---|---|---|---|
| **Projects with a preset** | 43,340 | 7 | 3,389 | 53% |
| **Project without a preset** | 115,500 | 37 | 2,735 | 87% |
| **All projects** | 158,840 | 17 | 6,124 | 64% |

(a) Enabled rules

| | **Total rules** | **Average** | **# projects** | **% projects** |
|---|---|---|---|---|
| **Projects with a preset** | 5,159 | 1 | 1,125 | 18% |
| **Project without a preset** | 20,122 | 6 | 1,601 | 51% |
| **All projects** | 25,281 | 3 | 2,726 | 29% |

(b) Warned rules

| | **Total rules** | **Average** | **# projects** | **% projects** |
|---|---|---|---|---|
| **Projects with a preset** | 17,263 | 3 | 3,395 | 53% |
| **Project without a preset** | 45,024 | 14 | 2,344 | 75% |
| **All projects** | 62,287 | 7 | 5,739 | 60% |

(c) Disabled rules

| | **Total rules** | **Average** | **# projects** | **% projects** |
|---|---|---|---|---|
| **Projects with a preset** | 65,762 | 10 | 4,465 | 70% |
| **Project without a preset** | 180,646 | 58 | 2,979 | 95% |
| **All projects** | 246,408 | 26 | 7,444 | 78% |

(d) All rules

Table 5.7: All used rules, divided into the different settings: enabled, disabled and warned rules. Each shows the total number of used rules, the average number of rules used per project, the number of projects that use at least one rule and finally the percentage of projects that use at least one rule.

rule. Furthermore, most rules are enabled on average per project for these two categories. However since these two categories also include the highest number of available rules (81 and 69), they are not the most commonly used categories in proportion with the amount of available rules. In that sense, Possible Errors is the most commonly used category with 24% of available rules being used on average per project, whereas with Stylistic Issues and Best Practices the ratio is 13% and 19%, respectively. The other three categories, Strict Mode, Node.js & CommonJS and ECMAScript 6, all have a similar number of projects with at least one rule, or around 28-29%.

The story is fairly similar when it comes to projects that do not use a preset. Much fewer rules are generally enabled as shown in the previous analysis on configuration prevalence, but the most popular categories remain the same. Stylistic Issues continues to be the most commonly used category, and even more so in this case where it is now proportionally more popular than the other categories, than it was when analyzing projects that do not use presets. Almost half of all projects that use a preset, or 47%, add at least one stylistic rule, and three rules are enabled on average per project. The second most commonly used

| **Category** | **Average** | **% available rules** | **# projects** | **% projects** |
|---|---|---|---|---|
| Possible Errors | 7.5 | 24.2% | 1,631 | 52.1% |
| Best Practices | 13.1 | 19.0% | 1,929 | 61.6% |
| Strict Mode[6] | 0.3 | 30.0% | 895 | 28.6% |
| Variables | 2.7 | 22.5% | 1,882 | 60.1% |
| Node.js & CommonJS | 0.9 | 9.0% | 917 | 29.3% |
| Stylistic Issues | 10.6 | 13.1% | 2,562 | 81.9% |
| ECMAScript 6 | 1.8 | 5.6% | 897 | 28.7% |

(a) Enabled rules per category for projects that **do not use a preset**

| **Category** | **Average** | **% available rules** | **# projects** | **% projects** |
|---|---|---|---|---|
| Possible Errors | 0.4 | 1.3% | 777 | 12.1% |
| Best Practices | 2.1 | 3.0% | 1,171 | 18.3% |
| Strict Mode | 0.1 | 10% | 448 | 7.0% |
| Variables | 0.4 | 3.3% | 1,043 | 16.3% |
| Node.js & CommonJS | 0.2 | 2.0% | 279 | 4.4% |
| Stylistic Issues | 3.0 | 3.7% | 3,020 | 47.1% |
| ECMAScript 6 | 0.6 | 1.9% | 799 | 12.5% |

(b) Enabled rules per category for projects that **do use a preset**

Table 5.8: **Enabled** rules per category, showing the average number of rules used per project, the average percentage of rules used per project out of all available rules in the category, the number of projects with at least one rule and the percentage of projects with at least one rule out of all projects. Categories are listed in the order as they are mentioned in the ESLint documentation [85].

category is again Best Practices, with 18% of projects specifying at least one rule and where two rules are used on average per project. An interesting aspect is that the distribution of the three less commonly used categories is notably different now than for projects that do not use presets. Before, more or less the same number of projects enabled a rule from these three categories, whereas now Node.js & CommonJS is by far the least used category with 4% and ECMAScript 6 having as many projects enabling a rule as Possible Errors, or around 12%.

**Disabled Categories**. When observing disabled categories for projects that do not use a preset, it is evident that the same two categories still remain the most commonly mentioned ones (Table 5.9). Stylistic Issues is the most commonly disabled category with 67% of projects disabling at least one rule and seven rules being disabled on average. The runner up is again Best Practices, with 54% of projects disabling a rule and with four rules being disabled on average. Interestingly, all other categories except for ECMAScript 6 have a

[6]The category Strict Mode is a special case in this analysis since it is the only category that includes only one rule (others have 10 or more rules). For that reason, it is less appropriate to report values for this category on the average percentage of used rule out of all available rules, as that percentage will always be relatively high compared to other categories. To maintain consistency, the values are reported in this table and in the following tables, but are not specifically analyzed in the text.

| Category | Average | % available rules | # projects | % projects |
|---|---|---|---|---|
| Possible Errors | 1.0 | 3.2% | 1,180 | 37.7% |
| Best Practices | 3.7 | 5.4% | 1,687 | 53.9% |
| Strict Mode | 0.3 | 30.0% | 980 | 31.3% |
| Variables | 0.8 | 6.7% | 1,111 | 35.5% |
| Node.js & CommonJS | 1.0 | 10.0% | 886 | 28.3% |
| Stylistic Issues | 6.8 | 8.4% | 2,096 | 67.0% |
| ECMAScript 6 | 0.9 | 2.8% | 532 | 17.0% |

(a) Disabled rules per category for projects that **do not use a preset**

| Category | Average | % available rules | # projects | % projects |
|---|---|---|---|---|
| Possible Errors | 0.3 | 1.0% | 1,569 | 24.5% |
| Best Practices | 0.6 | 0.9% | 1,505 | 23.5% |
| Strict Mode | 0.1 | 10.0% | 394 | 6.1% |
| Variables | 0.2 | 1.7% | 709 | 11.1% |
| Node.js & CommonJS | 0.1 | 1.0% | 404 | 6.3% |
| Stylistic Issues | 1.2 | 1.5% | 2,087 | 32.5% |
| ECMAScript 6 | 0.3 | 0.9% | 975 | 15.2% |

(b) Disabled rules per category for projects that **do use a preset**

Table 5.9: **Disabled** rules per category, showing the average number of rules used per project, the average percentage of rules used per project out of all available rules in the category, the number of projects with at least one rule and the percentage of projects with at least one rule

rather similar distribution, with 28-38% of projects disabling a rule. ECMAScript 6 therefore stands out with only 17% of projects disabling a rule from this category.

For projects that do use a preset, the distribution of disabled categories is somewhat different. Stylistic Issues continues to be the most commonly disabled category, with 33% of projects disabling at least one rule. The second category in this case is Possible Errors with 25% of projects disabling a rule. A singularity in these results is that the Variables category is by far the least disabled category out of the four popular ones with only 11% of projects disabling a rule. Even ECMAScript 6 is disabled more times, and is interestingly disabled much more now than the other two less commonly used categories, Strict Mode and Node.js & CommonJS.

**RQ$_{5A}$ Which are the most commonly used categories from ESLint?**
The category Stylistic Issues is enabled by the highest number of projects, followed by Best Practices and Variables, where almost half of all projects that use a preset enable at least one stylistic rule. Stylistic Issues is also the most commonly disabled category, followed by Best Practices and Possible Errors.

#### 5.3.2.2 Common Rules

Examining which categories are used gives a good overview of which types of rules are commonly used and what developers find important. In this part we dive deeper into these categories and examine which individual rules are enabled and disabled the most often.

**Enabled Rules**. It is shown in Table 5.10a that the three most commonly enabled rules for projects without presets, all belong to the Stylistic Issues category. These are formatting rules that enforce what kind of quotes should be used (`quotes`), whether semicolons should be placed at the end of a line (`semi`) and what kind of indentation should be used (`indent`). The next four rules in line are from the Best Practices category and Variables, namely `eqeqeq` which requires the use of the type-safe triple equality operator (=== instead of ==), `no-undef` which disallows undeclared variables, `no-unused-vars` which disallows unused variables and `curly` that enforces consistent use of curly braces for control statements. More rules in Best Practices follow, along with rules from the Possible Errors category and Stylistic Issues. The rule `no-dupe-keys` prevents errors when two keys in object literals are identical, `no-caller` disallows use of callers and callees which can otherwise make several code optimizations impossible and `no-unreachable` which disallows unreachable code, *e.g.*, after a `return`, `throw` or `break` statement. In general, all rules in this list belong to the four popular categories: Possible Errors, Best Practices, Variables and Stylistic Issues.

Table 5.10b shows the top 20 enabled rules for projects that do not use presets. Interestingly, these rules mostly belong to the Stylistic Issues category with 14 out of the 20 rules originating from the category. The same three stylistic rules as before are the most popular ones, followed by `linebreak-style` which enforces consistent line endings, `comma-dangle` which enforces or disallows trailing commas in object literals and `space-before-function-paren` which enforces consistent use of spaces before function parameter parentheses. These rules are followed by some of the few rules from other categories which were also detailed in the previous list.

**Disabled Rules**. For rules that are disabled in projects without presets, two rules stand out as the most disabled ones, as shown in Table 5.11a. The rule `no-underscore-dangle` from the Stylistic Issues category is disabled by nearly half of all projects that do not use a preset, where this rule disallows dangling underscores in identifiers which are commonly used to indicate a private variable. The other commonly disabled rule is `strict`, the only rule in the Strict Mode category, which requires or disallows strict mode directives. These two rules are followed with seven rules from the Stylistic Issues category, but it is the most commonly disabled category in this list. Included are rules that disallow ternary operators (`no-ternary`), require or disallow named function expressions for debugging purposes (`func-names`) and require constructor names to start with a capital letter (`new-cap`). The following rules belong to a set of more diverse categories, namely the four popular categories. These include `no-use-before-define` which disallows using variables before they are formally defined and `consistent-return` which enforces all `return` statements to always or never specify a return value.

Table 5.11b shows the most commonly disabled rules for projects that use a preset. The rule `no-console` from the Possible Errors category is by far the most commonly disabled

| | **Rule** | **Category** | **Frequency** | **% projects** |
|---|---|---|---|---|
| 1 | quotes | Stylistic Issues | 1,898 | 60.6% |
| 2 | semi | Stylistic Issues | 1,506 | 48.1% |
| 3 | indent | Stylistic Issues | 1,356 | 43.3% |
| 4 | eqeqeq | Best Practices | 1,338 | 42.7% |
| 5 | no-undef | Variables | 1,270 | 40.6% |
| 6 | no-unused-vars | Variables | 1,260 | 40.3% |
| 7 | curly | Best Practices | 1,232 | 39.4% |
| 8 | no-dupe-keys | Possible Errors | 1,228 | 39.2% |
| 9 | no-caller | Best Practices | 1,171 | 37.4% |
| 10 | no-unreachable | Possible Errors | 1,163 | 37.2% |
| 11 | no-eval | Best Practices | 1,155 | 36.9% |
| 12 | brace-style | Stylistic Issues | 1,126 | 36.0% |
| 13 | no-with | Best Practices | 1,123 | 35.9% |
| 14 | wrap-iife | Best Practices | 1,123 | 35.9% |
| 15 | no-irregular-whitespace | Possible Errors | 1,090 | 34.8% |
| 16 | comma-style | Stylistic Issues | 1,089 | 34.8% |
| 17 | no-cond-assign | Possible Errors | 1,085 | 34.7% |
| 18 | no-func-assign | Possible Errors | 1,083 | 34.6% |
| 19 | no-redeclare | Best Practices | 1,083 | 34.6% |
| 20 | no-invalid-regexp | Possible Errors | 1,079 | 34.5% |

(a) Top 20 enabled rules for projects that **do not use** a preset

| | **Rule** | **Category** | **Frequency** | **% projects** |
|---|---|---|---|---|
| 1 | semi | Stylistic Issues | 1,641 | 25.6% |
| 2 | indent | Stylistic Issues | 1,567 | 24.4% |
| 3 | quotes | Stylistic Issues | 1,336 | 20.8% |
| 4 | linebreak-style | Stylistic Issues | 888 | 13.8% |
| 5 | comma-dangle | Stylistic Issues | 714 | 11.1% |
| 6 | space-before-function-paren | Stylistic Issues | 605 | 9.4% |
| 7 | eqeqeq | Best Practices | 601 | 9.4% |
| 8 | no-unused-vars | Variables | 595 | 9.3% |
| 9 | curly | Best Practices | 515 | 8.0% |
| 10 | no-trailing-spaces | Stylistic Issues | 491 | 7.7% |
| 11 | brace-style | Stylistic Issues | 482 | 7.5% |
| 12 | strict | Strict Mode | 451 | 7.0% |
| 13 | space-before-blocks | Stylistic Issues | 424 | 6.6% |
| 14 | max-len | Stylistic Issues | 419 | 6.5% |
| 15 | no-use-before-define | Variables | 419 | 6.5% |
| 16 | keyword-spacing | Stylistic Issues | 416 | 6.5% |
| 17 | object-curly-spacing | Stylistic Issues | 396 | 6.2% |
| 18 | eol-last | Stylistic Issues | 386 | 6.0% |
| 19 | new-cap | Stylistic Issues | 366 | 5.7% |
| 20 | no-eval | Best Practices | 365 | 5.7% |

(b) Top 20 enabled rules for projects that **do use** a preset

Table 5.10: Top 20 enabled rules for projects that use and do not use a preset, showing the total number of projects using them, along with the percentage of those projects out of all projects that use or do not use a preset

rule, with 19% of projects specifying the rule. Logging with the console is mainly used for debugging purposes which should normally not be present in production code. Additionally, like before, `no-underscore-dangle` is also relatively commonly disallowed, or in 10% of all projects using a preset. Some rules that have not yet been discussed are `no-param-reassign` from the Best Practices category which disallows reassigning function parameters and `max-len` which enforces a maximum line length. Stylistic Issues is again the most commonly disabled category with half of the 20 rules belonging to the category.

**Enabled Rules Per Category**. We further examine in more detail the enabled rules in projects that do not use presets. Table 5.12 shows the 10 most commonly enabled rules for each of the ESLint categories for projects without presets. In the categories Possible Errors, Best Practices and Stylistic Issues, all top 10 rules are widely used where they are all enabled by at least 31% of projects without presets. For the Variables category, only a few rules are widely used (by more than 30%). For the other three categories, Strict Mode, Node.js & CommonJS and ECMAScript 6, all rules are used by less than 30% of the projects. As the categories Node.js & CommonJS and ECMAScript 6 did not make it to the top 20 lists in Table 5.10, none of their rules have been discussed. The most commonly enabled rule from the Node.js & CommonJS category is `handle-callback-err` which enforces handling of errors when using the callback pattern for asynchronous code. The second most commonly enabled rule is `no-new-require` which disallows immediate instantiation of required modules, as a slightly modified syntax of the creation can lead to errors. For the ECMAScript 6 category, the most commonly enabled rule is `no-const-assign` which disallows modifications to variables that are declared using `const` (instead of `let`). The second one is `arrow-spacing` which is a formatting rule that requires space before or after arrows in functions.

**Rarely Enabled Rules**. Lastly, there are several rules that are very rarely enabled in these 9,548 configuration files that were analyzed. For projects that do not use a preset (3,135 projects), 10 rules were enabled less than 10 times. These rules are shown in Table 5.13. Seven out of these 10 rules are from the Stylistic Issues category where the rules `sort-keys` and `capitalized-comments` are only used twice each. The former requires keys in objects to be alphabetically sorted and the latter enforces or disallows capitalization of the first letter of a comment. At the time of this text being written (June 2017), these rules have existed for 10 and six months respectively, whereas some of the oldest rules, such as `quotes` and `semi`, have existed for almost three years. Developers have therefore had less time to notice these newly created rules and furthermore, projects that are no longer active might never have had the chance to enable these. The creation time of the rules therefore plays an important part when analyzing rules that have not been enabled often. There are however two rules in this list that have existed for over a year which are `no-ternary` and `sort-imports`. In fact, `no-ternary` has been available for 2.5 years already, giving developers substantial time to include these in configurations if interested in doing so.

| | Rule | Category | Frequency | % projects |
|---|---|---|---|---|
| 1 | no-underscore-dangle | Stylistic Issues | 1,543 | 49.3% |
| 2 | strict | Strict Mode | 982 | 31.4% |
| 3 | no-ternary | Stylistic Issues | 686 | 21.9% |
| 4 | func-names | Stylistic Issues | 681 | 21.8% |
| 5 | new-cap | Stylistic Issues | 634 | 20.3% |
| 6 | one-var | Stylistic Issues | 608 | 19.4% |
| 7 | sort-vars | Stylistic Issues | 597 | 19.1% |
| 8 | padded-blocks | Stylistic Issues | 582 | 18.6% |
| 9 | no-use-before-define | Variables | 581 | 18.6% |
| 10 | consistent-return | Best Practices | 572 | 18.3% |
| 11 | no-console | Possible Errors | 570 | 18.2% |
| 12 | camelcase | Stylistic Issues | 565 | 18.1% |
| 13 | no-extra-parens | Possible Errors | 560 | 17.9% |
| 14 | func-style | Stylistic Issues | 549 | 17.5% |
| 15 | no-plusplus | Stylistic Issues | 535 | 17.1% |
| 16 | vars-on-top | Best Practices | 525 | 16.8% |
| 17 | no-shadow | Variables | 525 | 16.8% |
| 18 | no-warning-comments | Best Practices | 514 | 16.4% |
| 19 | valid-jsdoc | Possible Errors | 499 | 15.9% |
| 20 | wrap-regex | Stylistic Issues | 495 | 15.8% |

(a) Top 20 disabled rules for projects that **do not use** a preset

| | Rule | Category | Frequency | % projects |
|---|---|---|---|---|
| 1 | no-console | Possible Errors | 1,216 | 19.0% |
| 2 | no-underscore-dangle | Stylistic Issues | 646 | 10.1% |
| 3 | comma-dangle | Stylistic Issues | 479 | 7.5% |
| 4 | func-names | Stylistic Issues | 430 | 6.7% |
| 5 | no-param-reassign | Best Practices | 411 | 6.4% |
| 6 | strict | Strict Mode | 395 | 6.2% |
| 7 | no-use-before-define | Variables | 358 | 5.6% |
| 8 | consistent-return | Best Practices | 356 | 5.6% |
| 9 | max-len | Stylistic Issues | 315 | 4.9% |
| 10 | padded-blocks | Stylistic Issues | 305 | 4.8% |
| 11 | arrow-parens | ES6 | 266 | 4.1% |
| 12 | new-cap | Stylistic Issues | 262 | 4.1% |
| 13 | camelcase | Stylistic Issues | 259 | 4.0% |
| 14 | global-require | Node.js & CommonJS | 254 | 4.0% |
| 15 | no-shadow | Variables | 243 | 3.8% |
| 16 | arrow-body-style | ES6 | 224 | 3.5% |
| 17 | no-plusplus | Stylistic Issues | 204 | 3.2% |
| 18 | vars-on-top | Best Practices | 196 | 3.1% |
| 19 | space-before-function-paren | Stylistic Issues | 194 | 3.0% |
| 20 | id-length | Stylistic Issues | 193 | 3.0% |

(b) Top 20 disabled rules for projects that **do use** a preset

Table 5.11: Top 20 disabled rules for projects that use and do not use a preset, showing the total number of projects using them, along with the percentage of those projects out of all projects that use or do not use a preset

| | Rule | Freq. | % projects |
|---|---|---|---|
| 1 | no-dupe-keys | 1,228 | 39.2% |
| 2 | no-unreachable | 1,163 | 37.2% |
| 3 | no-irregular-whitespace | 1,090 | 34.8% |
| 4 | no-cond-assign | 1,085 | 34.7% |
| 5 | no-func-assign | 1,083 | 34.6% |
| 6 | no-invalid-regexp | 1,079 | 34.5% |
| 7 | no-ex-assign | 1,070 | 34.2% |
| 8 | use-isnan | 1,069 | 34.2% |
| 9 | no-obj-calls | 1,050 | 33.5% |
| 10 | valid-typeof | 1,034 | 33.0% |

(a) Possible Errors

| | Rule | Freq. | % projects |
|---|---|---|---|
| 1 | eqeqeq | 1,338 | 42.7% |
| 2 | curly | 1,232 | 39.4% |
| 3 | no-caller | 1,171 | 37.4% |
| 4 | no-eval | 1,155 | 36.9% |
| 5 | wrap-iife | 1,123 | 35.9% |
| 6 | no-with | 1,123 | 35.9% |
| 7 | no-redeclare | 1,083 | 34.6% |
| 8 | no-extend-native | 1,070 | 34.2% |
| 9 | no-implied-eval | 1,038 | 33.2% |
| 10 | no-new-wrappers | 1,028 | 32.8% |

(b) Best Practices

| | Rule | Freq. | % projects |
|---|---|---|---|
| 1 | no-undef | 1,270 | 40.6% |
| 2 | no-unused-vars | 1,260 | 40.3% |
| 3 | no-shadow-restricted | 1,030 | 32.9% |
| 4 | no-delete-var | 929 | 29.7% |
| 5 | no-use-before-define | 857 | 27.4% |
| 6 | no-undef-init | 809 | 25.8% |
| 7 | no-label-var | 804 | 25.7% |
| 8 | no-shadow | 638 | 20.4% |
| 9 | no-catch-shadow | 576 | 18.4% |
| 10 | no-undefined | 281 | 9.0% |

(c) Variables

| | Rule | Freq. | % projects |
|---|---|---|---|
| 1 | handle-callback-err | 657 | 21.0% |
| 2 | no-new-require | 589 | 18.8% |
| 3 | no-path-concat | 501 | 16.0% |
| 4 | no-process-exit | 304 | 9.7% |
| 5 | no-mixed-requires | 257 | 8.2% |
| 6 | callback-return | 192 | 6.1% |
| 7 | no-process-env | 139 | 4.4% |
| 8 | no-sync | 115 | 3.7% |
| 9 | global-require | 115 | 3.7% |
| 10 | no-restricted-modules | 66 | 2.1% |

(d) Node.js & CommonJS

| | Rule | Freq. | % projects |
|---|---|---|---|
| 1 | quotes | 1,898 | 60.6% |
| 2 | semi | 1,506 | 48.1% |
| 3 | indent | 1,356 | 43.3% |
| 4 | brace-style | 1,126 | 36.0% |
| 5 | comma-style | 1,089 | 34.8% |
| 6 | space-before-blocks | 1,045 | 33.4% |
| 7 | comma-dangle | 1,020 | 32.6% |
| 8 | space-infix-ops | 1,018 | 32.5% |
| 9 | eol-last | 995 | 31.8% |
| 10 | comma-spacing | 969 | 31.0% |

(e) Stylistic Issues

| | Rule | Freq. | % projects |
|---|---|---|---|
| 1 | no-const-assign | 477 | 15.2% |
| 2 | arrow-spacing | 452 | 14.4% |
| 3 | no-this-before-super | 450 | 14.4% |
| 4 | no-class-assign | 410 | 13.1% |
| 5 | constructor-super | 405 | 12.9% |
| 6 | generator-star-spacing | 401 | 12.8% |
| 7 | no-var | 401 | 12.8% |
| 8 | no-dupe-class-members | 337 | 10.8% |
| 9 | prefer-const | 255 | 8.1% |
| 10 | arrow-parens | 192 | 6.1% |

(f) ECMAScript 6

| | Rule | Freq. | % projects |
|---|---|---|---|
| 1 | strict | 900 | 28.8% |

(g) Strict Mode

Table 5.12: The 10 most commonly enabled rules for all categories and for projects that do not use a preset. Showing the total number of projects using them and the percentage of all projects using them.

| | Rule | Category | Frequency | % projects |
|---|---|---|---|---|
| 1 | sort-keys | Stylistic Issues | 2 | 0.1% |
| 2 | capitalized-comments | Stylistic Issues | 2 | 0.1% |
| 3 | prefer-destructuring | ES6 | 3 | 0.1% |
| 4 | max-lines | Stylistic Issues | 4 | 0.1% |
| 5 | multiline-ternary | Stylistic Issues | 4 | 0.1% |
| 6 | object-curly-newline | Stylistic Issues | 5 | 0.2% |
| 7 | line-comment-position | Stylistic Issues | 6 | 0.2% |
| 8 | require-await | Best Practices | 8 | 0.3% |
| 9 | no-ternary | Stylistic Issues | 8 | 0.3% |
| 10 | sort-imports | ES6 | 9 | 0.3% |

Table 5.13: The 10 least commonly enabled rules for projects without a preset

**RQ$_{5B}$ Which are the most commonly used rules from ESLint?**
The most commonly enabled rules are for code formatting and punctuation, *i.e.* to uphold a consistent use of quotes, semicolons and indentation. More stylistic rules are commonly enabled for projects that use presets. The most commonly disabled rules are for disallowing the use of logging to the console and the use of dangling underscores in variable names. A rule that disallows the use of the ternary operator is almost never enabled.

## 5.4 Discussion

### 5.4.1 Prevalence of Configurations

It seems that ESLint is in general configured quite extensively, where 78% of all projects specify at least one native ESLint rule and where a project uses 26 rules on average. Despite of this high frequency of configurations, it is also very common to use a preset where 67% of all projects include at least one.

**Presets**. The fact that such a high percentage of projects use a preset could indicate that developers like to take advice from others on which rules to use. Another explanation is that they try minimize the effort of using a linter by taking little time to create the configurations. Examining the presets that are used, it is interesting to see that only a few presets and plugins are by far the most commonly used ones. For the presets, there seems to be a wide agreement that `eslint:recommended`, `airbnb` and `standard` are suitable to use in JavaScript projects. It is perhaps possible to gain more insight into these high numbers by examining the types of rules that these presets include. The recommended settings from ESLint are said to include rules that report common problems [84] while the `airbnb` preset includes more diverse rules. More specifically, the number of rules that are included from each category for `eslint:recommended` and `airbnb` are shown in Table 5.14. The `eslint:recommended` mostly includes rules from the Possible Errors category, making it appealing for those that want to prevent bugs in the system. The `airbnb` preset is however much more versatile where it contains many rules from all categories. That preset might therefore be more appealing for those that do want to have rules for all these different aspects, but perhaps do not have the need to manually choose every single rule.

| | **eslint:recommended** | | **airbnb** | |
|---|---|---|---|---|
| **Category** | **#** | **%** | **#** | **%** |
| Possible Errors | 26+0 | 83.9% | 27+2 | 87.1% |
| Best Practices | 8+0 | 11.6% | 59+1 | 87.0% |
| Strict Mode | 0+0 | 0.0% | 1+0 | 100.0% |
| Variables | 3+0 | 25.0% | 9+0 | 75.0% |
| Node.js & CommonJS | 0+0 | 0.0% | 3+0 | 30.0% |
| Stylistic Issues | 1+0 | 1.2% | 51+1 | 64.2% |
| ECMAScript 6 | 7+0 | 21.9% | 27+0 | 84.4% |

Table 5.14: The number of rules enabled from each category in the two most popular presets, showing the number of rules in the format 'errors + warnings' and the percentage of used rules out of all available rules in category (only includes rules that were available in January 2017).

**Plugins**. Despite the fact that the main goal of ESLint was to create a pluggable linter, plugins are used less frequently than presets. They might however be underrepresented in this analysis as some presets already include plugins, such as `airbnb` that includes both the `react` and `import` plugins. In these cases, the plugins do not have to be specified in the configurations, unless they are to be modified. The most commonly used plugins can provide information on two different aspects: a) which JavaScript libraries are commonly used and/or need linting, and b) which features might be lacking from ESLint. Some of the most popular plugins are specifically written for external JavaScript libraries and tools such as `react`, `flow` [18] and `mocha` [23]. It is appropriate to keep a distance between ESLint and these types of plugins as developers use different types of libraries and tools in their applications. Other plugins however focus on native JavaScript features such as `import` [21] and `html` [19]. The `import` plugin relates to rules regarding import/export syntax for ES6 while the `html` plugin analyzes JavaScript code inside HTML files. The common usage of these plugins suggests that it might be appropriate to have these functionalities as a default part of ESLint.

**Prevalence of Rules When Using a Preset.** Regarding the prevalence of rules when presets are used, the proportion of projects that also modify individual rules is quite high. More specifically, 70% of these projects specify at least one native rule and use a number of 10 rules on average. It is therefore evident that relatively few projects use presets solely on their own, and instead add some custom configurations that fit better for the projects. It seems to be the case that developers rather feel that rules are missing from presets than feeling that some of the rules do not fit for their projects, as more rules are enabled on average than disabled. These numbers suggest that the presets might be incomplete or that they are unable to cover the diversity of all projects that apply them. It also suggests that developers have some personal preferences despite wanting to use a recommended set of configurations.

**Prevalence of Rules When Not Using a Preset.** For the prevalence of rules when presets are not used, it is curious to see that 95% of projects included at least one rule in their configurations. One would perhaps expect that ratio to be 100%, otherwise no rules are executed for the project. After examining a sample of the projects that fall under these 5%, they can be categorized in the following groups, ordered by frequency of occurrences: a) projects that only use plugins, b) projects that do not have any rule settings, c) projects with wrongly defined rules and d) projects with badly formatted code. There are several projects that solely use a plugin, either as a preset or specifying which individual rules form the plugins to use. These plugins are however not counted as actual presets since for those, we only examine the set of rules that the linter provides. Some projects do not have any settings at all whether it is rules, presets or plugins. These projects might therefore not actually be using the tool as no code is being analyzed, either intentionally or by mistake. However in some of these cases, another configuration file is specified such as via a command line script. In the third case, it was discovered during manual analysis that some projects define their rules wrongly, *e.g.*, having invalid settings to rules. Lastly, in rare cases, as described in the methodology chapter, the parser can miss out on some settings when it can not correctly parse a configuration file.

In general, a substantial amount of configurations is applied in the projects that are analyzed. We however only examine rules that are provided be ESLint, and not custom rules or rules added from plugins, which are instead simply ignored. If these rules would be analyzed as well, most numbers that have been reported would become somewhat higher as many projects also specify these custom rules. It is however also worth noting that for projects that use presets, an enabled rule can be one that was not originally included in the preset or it can be one that was already present in the preset and the more detailed settings of the rule are being reconfigured, *e.g.*, changing the indentation setting from being two spaces to four spaces.

### 5.4.2 Common Configurations

The categories Possible Errors, Best Practices, Variables and Stylistic Issues continue to be the main categories of interest. These four categories are most commonly enabled and disabled, both with and without using a preset.

**Enabled Categories.** Stylistic Issues is the most commonly enabled category, indicating that a consistent code style is the most important aspect in using a linter. This category is especially popular when compared to other categories for projects that use a preset. A reason for this might be that the same stylistic rules do not apply as commonly for all projects, whereas possible bugs in JavaScript systems might be more similar between projects. The stylistic rules are in nature more subjective so developers have different opinions on whether they should be turned on or off, and furthermore have different opinions on their more detailed settings when they are turned on, resulting in these rules being modified more often. Another possible explanation is the fact that the most commonly used preset, `eslint:recommended` mostly focuses on the Possible Errors category and has hardly any rules from the stylistic category. It might therefore be the case that projects using this preset are simply missing these rules from their configurations and thus add them. This reason would also explain why rules from the Possible Errors category are not commonly enabled when a preset is used. In general however, it is rather difficult to make assumptions about usage patterns for the projects that use a preset. Since presets can emphasize on very different types of rules, generalizing about any possible trends in the data becomes a tricky task.

For the Variables category, it is relatively common to enable at least one rule, but few rules are enabled on average. This might be purely due to the fact that there are much fewer available rules in this category compared to the other three popular ones. It could also mean that only very few rules from this category are thought to be important, but that those particular rules are in fact very important. Lastly, the category ECMAScript 6 is enabled by relatively many projects that use a preset. That might mean that presets do not have enough focus on ES6 or that developers feel that ES6 language features are in particular need of linting.

**Disabled Categories.** It is more difficult to make assumptions for categories that are commonly disabled than those that are enabled. For projects that do not use a preset, it is possible to explicitly turn off a rule, but not even specifying the rule has the same effect. There could be different reasons as to why developers choose to include disabled rules, perhaps for everyone to see which rules are available or possibly to explicitly state that some specific rules should never be turned on. It is however clear that Stylistic Issues is the most commonly disabled category, both for projects that use a preset and those that do not. This makes the category very controversial, as it is also the most commonly enabled one. That might not come as a surprise as we have seen that developers tend to have very different opinions regarding style related rules. This controversy in the data applies as well for the Best Practices category, but to a lesser extent.

**Enabled Rules**. Upon examination of the most commonly enabled rules, they seem to mostly have to do with either clear and consistent code (mainly formatting) on one hand, and possible bugs on the other. Various Stylistic rules are however more common in the top 20 list of enabled rules for projects that use a preset rather than for projects that do not. This could indicate that presets are lacking stylistic rules or that the settings to stylistic rules need more modification. Three rules stand out as being the most commonly enabled ones for all projects which are `quotes`, `semi` and `indent`, who all have to do with different code formatting. According to the interviewees, when there are inconsistencies with these punctuations, the code becomes more difficult to read and can result in the code being misunderstood. Interestingly, these rules are not included in the `eslint:recommended` preset and in fact, 11 of the top 20 rules for projects without presets are not included in the preset. For projects that do use a preset, only two rules out of the top 20 enabled rules are present in the `eslint:recommended` preset. However, all rules in both top 20 enabled lists are present in the `airbnb` preset. The `airbnb` preset therefore seems to capture the rules that developers find most important to use. However, as we do not know if these rules are being added on top of the presets or being reconfigured, it might also mean that the detailed settings to these rules in the `airbnb` preset are not appropriate for all projects.

On a somewhat different topic, it is also worth examining which rules are seldom enabled. We saw that two rules were enabled in only 0.3% of all projects without a preset, even though they had been available for a substantial time. While it is good to have a wide variety of rules to choose from, the high number of available rules can also make it more difficult to configure the tool as there are many rules to shift through and decide on. This list could therefore be simplified by removing the rules that are hardly ever used by developers, after being available for a certain amount of time, such as one year.

**Disabled Rules**. For the most commonly disabled rules, just over half of the top 20 rules for both projects with and without presets, belong to the Stylistic Issues category. Interestingly, only four rules originate from the Possible Errors category, thereof only one rule for projects that use a preset. That particular rule is however by far the most disabled rule for projects with presets, namely `no-console`. As reported in some of the interviews, it can be frustrating to have some rules like `no-console` turned on when the code is still in development and to have a build interrupted because of them. However, these instances might not be wanted in the code when it goes to production. Another rule that stands out is `no-underscore-dangle`, which indicates that developers do want to use the tradition of naming private variables with an underscore as a prefix, or at least to make some kind of distinction between variables with this naming tradition. Regarding the commonly disabled rules that are enabled in the popular presets, only the rule `no-console` is included in the `eslint:recommended` preset. However for `airbnb`, 13 out of the top 20 most disabled rules for projects that do not use presets, are present in the configurations, and 19 out of the top disabled rules for projects that do use a preset. Perhaps it would therefore be better to remove some of these rules from this popular preset, as they are commonly being disabled. Some rules have however proved to be controversial, in the sense that they are often both enabled and disabled. These rules include `strict`, `comma-dangle`, `space-before-function-paren`, `max-len`, `no-use-before-define` and `new-cap`.

## 5.5 Conclusion

We studied how developers configure ESLint by analyzing the configuration files of 9,548 JavaScript projects on GitHub. A tool was created for this purpose which was made publicly available. The majority of these projects use a preset in their configurations with a few rules added, removed or modified. Only a handful of presets and plugins are very popular amongst the projects. Furthermore, the projects that do not use a preset generally specify many rules in their configurations. Lastly, it is most common to apply rules that aid in maintaining code consistency and especially regarding punctuation in code.

# Chapter 6

# The Experiences and Perceptions of the JavaScript Community

The previous two chapters reported on detailed qualitative data on developers' perceptions of using ESLint on the one hand, and a large amount of quantitative data about how the tool is configured on the other. In this chapter we challenge our previous findings to generalize our original results from conducting interviews and to further explain the results obtained by mining configuration files on GitHub. To that goal, we surveyed 337 developers from the JavaScript community about their perceptions and experiences with using linters. The survey was built upon the previously acquired knowledge where the information is used as input into both open and closed questions. The topics of this chapter are therefore somewhat repetitive of previous chapters, but with a completely different approach of obtaining the data, where the results are used to strengthen our findings and the implications we present in the final chapter of this thesis. Although the survey is mostly focused on the usage of ESLint, we also obtain new insights into the usage of other linters. We now had the chance see why developers use any linter and the challenges that they generally entail. Furthermore, we present reasons why some linters are considered to be better than others, giving tool makers valuable information on how to improve linters in the future.

## 6.1 Research Questions

We further examine the topics that have been discussed in the previous two chapters where we now analyze the experiences of a large diverse group of JavaScript developers. More specifically, we examine the following:

**RQ$_6$ Why do developers use linters and what are the most important features?**
We examine why developers generally want to use a linter and which aspects of the code they wish to improve. We also study why some developers chose not to use a linter for their JavaScript projects to see which aspects of linters can be improved. Furthermore, we see which linters are the most preferred ones and why, and with that discover which are the most important features for linters to have.

**RQ$_7$ How do developers configure linters?**
We investigate what methods JavaScript developers use to choose rules for a project and further see whether they prefer to apply custom configurations or to use presets.

**RQ$_8$ Which types of rules are important to use for a linter?**
We collect the opinions of the developers on which types of rules are important to include in configurations for a linter. We specifically examine which ESLint categories are the most important ones to use and also which specific rules within the four main categories.

**RQ$_9$ What is challenging about using a linter?**
We study the challenges that developers face while using a linter, to learn what can be improved to make their experience with using a linter better. Additionally, we specifically examine how developers perceive false positives while using a linter.

## 6.2 Methodology

There are several aspects to consider when conducting a survey in order to reach the goals of a study and to obtain valid results. It needs to be carefully designed, tested and reviewed before being sent out to the public. The participants of the survey then need to be selected so that they can represent the target population of the study. Finally, the data needs to be thoroughly analyzed and reported in an appropriate way.

This survey was constructed under guidelines from both social sciences and software engineering research. The main resources that were used are from Fink [87], De Vaus [81] and Kitchenham [113], who all provide strategies and advice on conducting surveys. The tool SurveyGizmo [55] was used to facilitate the survey, since it is a well equipped online survey tool and was easily accessible to the author. A PDF version of the complete survey is available online [154]. Before presenting the results of the survey, we describe how the survey was designed and evaluated along with how sampling was performed and how the data was analyzed.

### 6.2.1 Survey Design

Questionnaires need to be carefully designed as it is not easy to change them after they have been published or to ask participants further questions after they have completed them [81]. It is therefore vital to design the survey in a cautious manner to make sure to (correctly) obtain all information one seeks after. In the following we describe how the survey and individual questions were designed.

#### 6.2.1.1 Designing Questions

There are multiple different types of questions that can be used in a questionnaire. Open questions allow for unanticipated answers but can also be challenging for participants to complete [87]. Closed questions are easier to answer and can provide more reliable results, but need to be well thought out so that options are both appropriate and exhaustive [87, 81, 114]. For this questionnaire, it was decided to mainly take advantage of closed questions in order to make the survey more reliable and more compelling to participants to complete. Four types of questions were used in total; open questions, categorical questions (multiple choice and checklist) and ordinal questions using a Likert scale. Not only did these different types of questions serve our purpose in the best way, but also provide the participant with more variety of question formats, thus making the survey more interesting to complete [81]. To minimize the risk of forcing opinions on participants, each closed question, where appropriate, contained an option where the user could write his or her own response, along with having a neutral option in ordinal questions. Moreover, to minimize the risk of bias due to the order of options in the survey, the options were randomly shuffled for each participant, wherever appropriate.

To choose the options for the closed questions, input was used from the results of the previous two chapters. As an example, when asking why developers use a linter, the options included the seven reasons that were identified in the interviews (Chapter 4), along with three open options where the participant could enter his or her own reasons. A second example, when asking which presets developers like to use, the five most commonly used presets from the repository mining (Chapter 5) were used as options, again also with three open options.

Questions also need to be carefully written so that their intention is clear to all participants [87, 81, 114]. Special care was taken as to make the questions straightforward, unambiguous and to address only one item at a time. If questions are ambiguous or confusing in any way, participants might understand them differently and provide inconsistent answers, thus compromising the validity of the survey. It was also taken into consideration to use correct grammar, apply appropriate language and for the questions to not be too lengthy. Additionally, biasing or leading words were avoided along with any negative phrasing.

#### 6.2.1.2 Designing the Complete Survey

There are many things to consider when trying to make a survey appealing and interesting enough for potential participants to both start and complete it. For this purpose, a motivational cover letter was placed in the beginning of the survey [87, 81, 134, 150]. This short letter explains the purpose of the study and the possible benefits it can have for developers and tool makers. Furthermore, it was explained that the results would only be used anonymously but that it was also possible to leave behind an email address in order to receive the results when the study is completed. Lastly, as monetary prizes can increase response rates [145], it was announced that one lucky participant would receive an Amazon gift card worth 50 dollars.

Another thing to consider is the flow and order of questions. The survey was constructed so that it started with easy questions, ended with the most time consuming questions and where relevant questions were grouped together into separate pages [81, 134, 115]. It is debatable whether demographic questions should be placed in the beginning of a survey or in the end, but it was decided to start with them in order to have easier questions in the beginning, hoping to warm up the participants and get them engaged in the survey

[81, 112]. These demographic questions included the participant's gender and country of residence along with years of experience as a developer and the nature of his or her work. These questions were included to later analyze the diversity of the study sample.

The survey was structured so that it is divided in four different sections which correspond to the four research questions of this chapter, in addition to the first section that contains the demographical questions about the participant. The sections are mixed with questions that ask about information received in the interviews on one hand and in the repository mining on the other. Choosing which options to include in questions was most often a straight-forward task, where either all reported values from the interviews were used or the most common configuration patterns that were found with the repository mining. However for $RQ_8$, where we ask which types of rules are important to use in configurations for four different categories, a specific method was needed to choose which rules to include as options as there are too many rules available to ask about. A set of 14 rules was therefore constructed for each category, except for Variables where there are only 12 available rules. These sets were created with the following criteria: First we select the five most commonly enabled rules without using a preset and five with using a preset. Some rules can appear inn both these sets where additional rules are then chosen. If an even number of additional rules were needed, one rule was added from each list of commonly enabled rules (*e.g.*, the sixth top enabled rule without a preset and also the sixth top enabled rule with using a preset). If an odd number of additional rules were needed, the next rule in line which has been more commonly enabled is added to the set. The same process was repeated with the most commonly disabled rules, except only four rules were added, consisting of two rules where a preset is used and two where no preset is used. The point of this system was mainly to see whether the most commonly used rules are also explicitly considered important by developers. However, the commonly disabled rules were also included so it would be possible to see a difference in rating and to evaluate whether the disabled rules are in fact less important. Like with many other questions, the order of the rules was randomized, along with the order of the questions themselves as they appeared for each category. Finally, the scale to rate the importance was designed as the following: a) Unimportant, b) Slightly important, c) Neutral/Not applicable, d) Important and e) Very important.

Not all questions were applicable for every single participant. Early on, participants were asked if they had ever used a linter in a project. If the answer to that question was negative, they were only presented with two additional questions, asking how important they consider it to be to use a linter and why they have never used a linter. Other participants received questions on the usage of linters, including a question asking which linters they had used. If a participant had not used ESLint in any project, he or she was not presented with questions that had specifically to do with the rule selection of that linter. Instead they (and also those that had used ESLint) only received more generic questions about why they use a linter, what methods they use to configure it and which challenges they have faced.

### 6.2.2 Survey Evaluation

It is crucial to evaluate the survey before publishing it in order to make sure that everything works as intended. The purpose of evaluating the survey is to expose its possible weaknesses, such as whether questions can be interpreted in different ways or if participants get too bored with the survey and therefore do not complete it [81, 116, 112]. The most common types of evaluation methods for survey instrumentations are focus groups and pilot studies [124, 116]. In the following sections we describe how the survey was evaluated and what key improvements were made in the process.

#### 6.2.2.1 Evaluation method

The evaluation approach taken for this survey was to perform pilot studies but rather in the nature of moderating focus groups. More specifically, six participants with different characteristics were recruited to take the survey. A session was held with each participant individually where they were asked to take the survey and describe their thoughts while doing so. Two sessions were performed in person while the other four were conducted over a video call on Google Hangouts where the participant shared his screen while taking the survey. The participants were asked to read each question out loud and to express their thoughts and understanding of the questions. They were especially encouraged to speak out if they thought any questions were unclear or vague and to express whether answer options were appropriate. Moreover, they were asked to be completely honest and to answer the questions as they would under normal circumstances. This last request was asked to try to see how a typical participant would experience the survey, instead of a colleague that might answer the questions with more care as a favor to the researcher. Furthermore, before answering the questions, they were asked to read the introduction text to evaluate how appropriate and motivating it was.

A checklist of specific elements to look out for was created before conducting the pilot tests [87, 81, 112, 116, 124]. The following items were considered in each pilot study, starting with the elements that relate to the individual questions of the survey:

1. Is the question clear and non ambiguous to the participant?
2. How does the participant understand the question?
3. Why did the participant answer the question in this way?
4. Are the answer options appropriate and exhaustive?
5. Are all terms understood?
6. Does the question seem relevant to the participant?
7. Is the question leading in any way?

For the survey as a whole, the following items were taken into consideration:

1. Is the participant motivated to take the survey?
2. Is the length of the survey appropriate?
3. Is the grouping and order of questions natural and appropriate?

4. Are the instructions clear and navigation easy?
5. Is the participant bored with the survey?
6. Is the participant overall satisfied with the survey?

It was possible to answer some of these questions based on the participants' expressions but in other cases they were explicitly asked about these items. Additionally to these pilot tests being performed, the instrument was carefully tested by the author to ascertain that it was working correctly, *e.g.*, displaying correct questions based on previous answers.

The six participants for the pilot tests were friends and colleagues of the author that were known to have used JavaScript. The intention was to test the complete survey a few times which required participants that had used ESLint. It was also intended to test all three paths of the survey, leading to other participants being recruited that had never used a linter and that had used a linter but not ESLint. Further information about the participants is shown in Table 6.1. To meet all these requirements, it was necessary to perform at least four tests. A number of six participants were however recruited to be able to obtain more feedback. After performing six pilot tests, we were confident that the survey was in a good state to be published.

| | **Residence** | **Role** | **Experience** | **Main language** | **Used a linter** | **Used ESLint** |
|---|---|---|---|---|---|---|
| 1 | Netherlands | Student | 3 | Yes | Yes | No |
| 2 | Iceland | Developer | 4 | Yes | Yes | Yes |
| 3 | Mexico | Team leader | 1 | Yes | Yes | Yes |
| 4 | Greece | Pen tester | 1 | No | No | No |
| 5 | Netherlands | Developer | 1 | Yes | Yes | No |
| 6 | Sweden | Developer | 1 | Yes | Yes | Yes |

Table 6.1: Participants in the pilot study, in the order of tests performed. The columns show the participants' country of residence, their primary role in software development, years of experience with using JavaScript, whether JavaScript is their main programming language, whether they have used a linter for JavaScript and whether they have used ESLint.

#### 6.2.2.2 Improvements

The pilot study proved to be extremely useful where several improvements were made based on the participants' feedback. After conducting each pilot study, the survey was modified based on the their comments. Each participant therefore received an improved version of the survey, allowing them to focus on other aspects than the previous participant. The key improvements were the following:

- **Participant 1**
  1) The contributions of the study were emphasized in the introduction letter to provide more motivation for the participant. 2) Explanations were added for the terms *preset* and *false positive* for those that might not recognize them. 3) Three questions were

rephrased so that it would be obvious for the participants to answer them based on their previous experience, and not opinions they might agree to.

- **Participant 2**
  1) The neutral options of all Likert scales were moved from the far right to the middle, in order to make the process of answering more intuitive for the participant. 2) When asking about *presets*, explicit instructions were added to not confuse it with the term *plugin*. 3) All URLs were complemented with a hyperlink to make it easier for the participant to access them.

- **Participant 3**
  1) For all questions where participants are asked to rate the importance of ESLint rules, links to the documentation of each rule were added, to make it easier for the participant to understand the rules. 2) The same questions were divided into individual pages in the survey, so that the participant rather feels that he or she is progressing after finishing each question.

- **Participant 4**
  1) When the participant is asked to enter his or her email address, a sentence was added to emphasize that the results would be used anonymously and that the email would not be distributed to any third parties. This was done so that the participant would feel safer to enter the email address.

- **Participant 5**
  1) For texts that the participant is less likely to read, important keywords were written in bold to make them more easily noticeable. 2) The options for a question regarding false positives were changed from a five point Likert scale to a seven point Likert scale, to better capture the respondents' experience.

- **Participant 6**
  1) When asking about presets, a distinction was made for those that are available as both presets and individual tools (such as Standard), to remove possible ambiguity. 2) All links were made to open in new tabs in the browser, to not interrupt the survey taking. 3) Essay questions were added alongside several closed questions where participants could optionally explain their choices, thus providing additional insight for the researcher.

Additionally, small incremental changes were done to the motivation letter to make it as appealing but also as concise as possible. Interestingly, there was a wide agreement with offering a monetary prize where it was considered to be very motivating for possible respondents. Finally, when all pilot tests were completed, all responses were examined to identify any further possible problems. One problem was identified in that process where an essay question, about why developers use a specific linter, was modified since it was not clear to which linter the responses were referring to. Instructions were therefore added to the question to explicitly state which linter the participants are discussing.

### 6.2.3 Sampling and Responses

When conducting a survey one needs to decide who the *target population* is and from that group a representable *sample*, as it is not possible to survey the whole population [87, 117]. The external validity of the results obtained with a survey much depend on this participation sample [117, 124]. There are multiple ways to obtain this subset of people with both *probabilistic* and *non-probabilistic* sampling methods. Probabilistic methods are better suited to ensure a valid sample for which the results can be generalized for the whole population. These methods however require a previously known set of possible participants which is often unknown in software engineering research [150, 80]. For that reason, a common approach in software engineering research is to apply *convenience sampling* which relies on people who are available and interested to take part in the survey, although with a risk of jeopardizing the generalizability of the results [150, 117, 87]. Since population sources are often not available, software engineering researchers rely on recruiting participants from other resources such as from the web, from conferences or by approaching colleagues [150, 80].

In this study the target population is all JavaScript developers. The survey is specifically directed towards those that have used a linter and in particular those that have used ESLint, as they can answer more of the survey's questions and have more knowledge on the topic. Unfortunately there does not exist a list of all JavaScript developers in the world so it is not possible to apply probabilistic methods to obtain the sample. Instead, convenience sampling was applied where the survey was advertised in several places on the web where it was likely to find members of the target population. More specifically, the survey was distributed in the following four locations (the references point to the direct post where the survey was promoted):

1. **JS.is**. An Icelandic JavaScript user group on Facebook with 789 members [156][1].
2. **JS reddit**. A subreddit about JavaScript on the popular community-driven social news site reddit.com with 111,649 subscribers [157].
3. **Echo JS**. A community-driven news site about JavaScript development [7][2].
4. **Twitter**. A news and social networking site [158].

The survey was originally distributed on JS.is to start off slowly as it is a relatively small community. It was also intended as the final test of the survey, where the replies were examined to identify any possible remaining problems (no such problems were detected). A post was created on the site with a motivational text to take the survey, both in English and in Icelandic. Similar to the opening letter of the survey, the text was carefully constructed to appeal to the target audience and to persuade them to participate in the survey. The post received 27 "likes" and 15 full survey responses[3].

[1]All information about the four resources was collected on June 14th 2017.

[2]This reference does not contain a direct link to the promotion post as it is not made accessible by the Echo JS website.

[3]Here we only report on numbers of full responses, as opposed to partial responses.

After verifying these first responses, the survey was posted on the popular social news site Reddit on a special page dedicated to JavaScript. The same motivational text was posted on this site (now only in English), with the title "*Do you want to know how developers use and configure JS linters? take this survey!*". The post quickly became the most popular one on the site for a limited time, remaining in first place for at least 5 hours. After the first 24 hours the post declined to the 9th place and finally disappeared from the first page after two days. The post was viewed by 1,300 readers, received a number of 90 "upvotes" and several comments such as: "*Very nice. Please post the paper and results when finished!*". The highest number of full responses was collected from this resource, or 268 in total.

The third distribution place was the Echo JS community, where only a link to the survey was posted without any accompanying text, except for the following title: "*Ever used a JS linter and maybe struggled with the configurations? take this survey!*"[4]. The post peaked in popularity in the third place where it stayed for around four hours, then declined gradually down the list and eventually disappeared from the front page after five days. This community appears to be much smaller and less active than on Reddit, where the post received merely four upvotes, no comments and a number of 44 full survey responses.

The last distribution place was the popular social networking site Twitter. A "tweet" was posted on the author's personal Twitter account which was then "retweeted" by 13 other individuals. The tweet message was "*Ever used a JS linter and maybe struggled with the configurations? Want to know how others do it? take this survey!*". Even though five of the "retweeters" had over 1,300 followers (and up to 5,142), the tweet attracted few engagements as shown in Table 6.2, where only 28 people opened the link to the survey. Eventually, only 10 full responses were obtained from participants that were directed from Twitter.

| **Activity** | **Frequency** |
|---|---:|
| Impressions | 4,615 |
| Profile clicks | 41 |
| Link clicks | 28 |
| Detail expands | 23 |
| Retweets | 13 |
| Likes | 9 |
| Follows | 1 |
| **Total engagements** | **115** |

Table 6.2: Engagements in the tweet that was used to promote the survey according to Twitter

Distributing the survey in these four locations resulted in a total number of 337 completed responses, as shown in Table 6.3. Furthermore, additional 476 partial responses were received (with no information available on the different sources), which range from having no answers to being almost complete. The completion rate is therefore 42.0%. The survey

[4]It was possible to create a post with a text description, similarly as on Reddit, but as all other then current posts only referred directly to links, it was decided to follow the dominating trend.

was first posted on the web on June 1st and eventually closed on June 14th, thus being available for 14 days.

| **Location** | **Responses** |
|---|---:|
| JS.is | 15 |
| JS Reddit | 268 |
| Echo JS | 44 |
| Twitter | 10 |
| **Total** | **337** |

Table 6.3: Number of full responses from each location where the survey was distributed

### 6.2.4 Data Analysis

The tool that was used for conducting the survey, SurveyGizmo, offers the functionality of automatically creating reports out of all or selected responses. The responses can be filtered based on their status (completed, partial or disqualified in the case for pilot study answers) or based on answers to selected questions. This report proved to be sufficient to analyze all closed questions in the survey, as they also provide features to display statistical information such as the average for nominal responses.

For the open questions, further manual analysis was needed. There are eight questions in the survey that are completely open (essay questions) but many more that are closed but additionally contain an open answer possibility to add an extra option. All these answers need to be analyzed with a qualitative method. Similarly as when processing the interview in Chapter 4, an inductive method [87] was used where the main themes of each question were identified as they emerged when processing the answers. Each identified category received a code and each answer was labeled with at least one code. When needed, the coding was conducted on different levels of detail, where broad categories were identified which were then sub-classed into more detailed ones [81]. In cases where answers described more than one item, *e.g.*, listing several different reasons for something, it was decided to give multiple codes instead of choosing only one and then possibly disregarding parts of an answer.

### 6.2.5 Limitations

#### 6.2.5.1 Survey Design

It is important to consider both the *validity* and *reliability* of a survey [87, 81]. The validity mainly has to do with how well surveys measure what they are intended to measure, while reliability considers how consistent and true the responses are. To try to ensure that this survey was both valid and reliable, much consideration went into writing relevant questions that use direct and appropriate language. All questions were then reviewed and evaluated in pilot tests with members of the target population. To further evaluate the reliability of the survey, the responses were reviewed in separate batches based on the source of the participants, *e.g.*, a separate batch for all participants that entered the survey via the promotion on Reddit. No major discrepancies could be identified between the different groups, conforming the *inter-rater reliability* of the survey.

#### 6.2.5.2 Survey Evaluation

The survey was pilot tested in order to identify possible problems and to make general improvements. Six participants were recruited who provided extremely valuable feedback for the survey where some crucial improvements were made. After these six tests, the author was confident that these improvements were sufficient for the survey to be considered both valid and reliable. It is however possible that not all problems were detected in these tests and that more tests would have led to making more improvements. To make the pilot tests as efficient and effective as possible, several sources of literature were examined to learn about what aspects to focus on. This preparation played a key part in the resulting confidence that the author had about the final quality of the survey. Additionally, when examining the final responses of the survey, no major problems could be identified with the answers.

#### 6.2.5.3 Sampling and Responses

It is important to consider the response rate and the representativeness of a survey's response set [118], but it is however difficult to calculate a response rate for this kind of convenience sampling. For two of the distribution places, JS.is and Echo JS, it is impossible to know how many individuals viewed the post to estimate a response rate. For JS Reddit, it is known that the post was opened by 1,300 individuals, resulting in a response rate of 20.6% for complete responses[5]. It is however not known how many people saw the post on the front page and did not decide to take a further look, consequently not having access to the survey link. For the last location, Twitter, information is provided on how many users were exposed to the tweet on their timeline (number of impressions in Table 6.2). The tweet reached a number of 4,615 users but it is difficult to predict how many of those users actually noticed or read the tweet.

Perhaps more important in this case is the representativeness of the response set. Three of the locations (JS.is, JS Reddit and Echo JS) are communities that are specifically created for JavaScript enthusiasts, which is exactly the target group of the study. Furthermore, as the title and description of the survey refer to linters for JavaScript, it should attract mostly JavaScript developers that have used a linter in the past. For Twitter, all retweeters except for one (who does not have a self-description) are either developers or software engineering professors or researchers. These people most likely have many followers of related professions where however not all might be involved with JavaScript. All in all, the response set is considered to be a representative sample. The representativeness can be evaluated even further in a later section where we analyze the results from the survey

[5]It is estimated that 378 partial responses were received from Reddit, resulting in a response rate of 49.7% for all responses.

which includes information on the participants' experience in software development and with JavaScript.

#### 6.2.5.4 Data Analysis

As before in Chapter 4, a possible limitation is regarding the quality of the coding process for qualitative answers. The codes were created solely by the author, based on her interpretation of the answers and her knowledge on the topic. This interpretation and classification might however be performed differently by another person. Due to this concern, the author tried to not over-interpret any answers that were not completely obvious in meaning, and rather assign fewer codes. To further mitigate this issue, the codes are available for anyone to review [152]. Additionally, to try to prevent any coding errors [81], all answers and their accompanying codes were reviewed and verified after completing the coding process.

## 6.3 Results

As reported before, we obtained a number of 813 responses, with 337 completed and 476 partial responses. In this analysis we only examine the completed responses as 1) the quantity of responses was considered to be sufficiently large and 2) earlier questions such as demographical questions were more frequently answered in partial responses where the results of those questions would not represent the sample that answered all other questions. However, for the completed responses, it was still possible to skip any question. It was uncommon for participants to skip closed questions, but very common with some of the open ended questions, with some of them being intentionally worded to appear optional. These particular questions were intended to provide the participants with an opportunity to explain their choices in the closed questions if they wished.

In the following sections, the results of the survey are reported, starting with background information on the participants. To make interpretation of the data easier for the reader in some cases, graphs are displayed that were created with the online survey tool.

### 6.3.1 Participants

Several demographical attributes were collected about the participants, namely their gender, country of residence and several items related to their experience with the topics discussed in this survey. These latter questions were intended to measure the representativeness of the sample and also to decide which further questions to display to the participants, as explained earlier.

Nearly all of the survey's participants are male, or 96.4%, with only four female participants, accounting for 1.2%. One participant chose Gender Variant/Non-conforming while the rest preferred not to answer the question. The respondents' country of residence covers 53 different countries where more than half of all participants, or 69.7%, come from seven dominating countries as shown in Figure 6.1, most commonly from the United States.

A large majority of the participants identified their primary role in software development as a developer, or 86.1%, as shown in Figure 6.2. Other noticeable roles are team leader

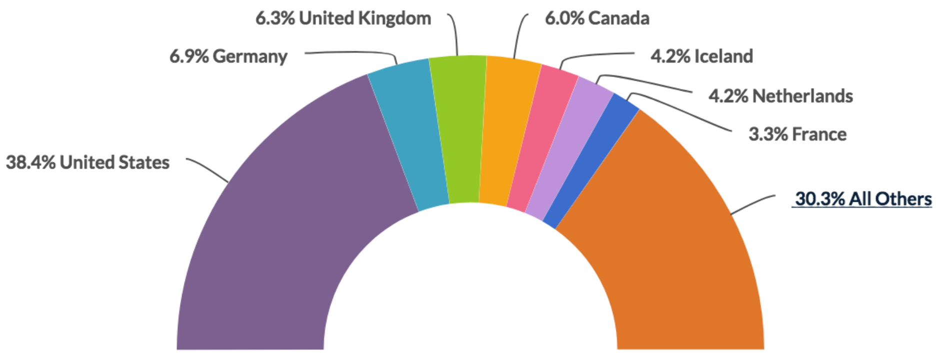


Figure 6.1: The country of residence of participants

(6.3%) and student (4.8%). The experience of the participants in software development and with JavaScript is shown in Figure 6.3. This was an open question where a number was required, resulting in many different answers. The average experience of the participants in software development is 8.0 years, and their average experience in working with JavaScript is 5.8 years. All participants claimed to have used JavaScript in the last year and 96.1% reported that it was one of their main programming languages. They were further asked in which field they regularly work when using JavaScript. The majority regularly works with commercial software (87.0%) while around half of the participants work with open source software (50.6%) (some participants chose both options). Finally, 93.7% claimed to have used a linter in a JavaScript project. The 21 participants that had never used a linter were not presented with the greater part of the rest of the survey.

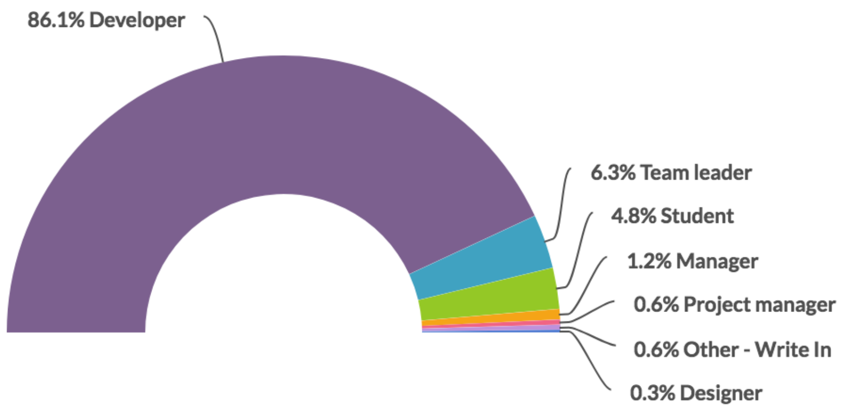
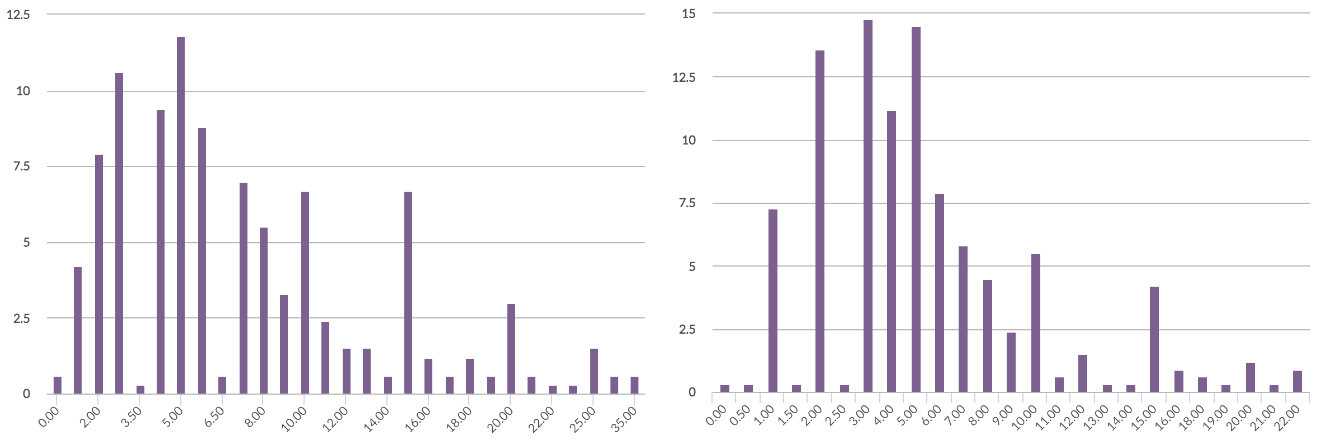


Figure 6.2: The primary roles of participants in software development

### 6.3.2 RQ$_6$ Why Developers Use Linters

In this section we examine why developers use a linter and why a few participants had never used a linter. Additionally we study which linters the developers have used and see why they



(a) Experience in software development

(b) Experience with using JavaScript

Figure 6.3: Participants' experience in software development and with JavaScript. Axes show years of experience and percentage of participants with the corresponding answer.

prefer to use one linter over another.

#### 6.3.2.1 Reasons for Using a Linter

Participants were asked why they had used a linter in their previous experience where they were presented with seven different reasons which were obtained while conducting the interviews with the 15 experts. Table 6.4 presents the results on how the participants agreed with the proposed reasons[6]. It is evident that most participants agree with two reasons, maintaining code consistency and catching bugs and typing mistakes. There is also a wide agreement on the reason that it saves time that goes into discussing code style. A general agreement, but not as definite, is with the reasons of avoiding ambiguous or complex code and with automating parts of code reviews. The only two reasons that have a significant amount of disagreement with is avoiding negative comments in code reviews (and thus sparing the developers' feelings) and learning new JavaScript features or syntax. Still, 26.5% and 30.7% of the participants, respectively, agree with these two reasons, thus involving more disharmony with the answers than for the previous reasons.

Additional 38 reasons were provided by the participants with some of them being repetitions of previously provided reasons. The most commonly mentioned observation was to enforce coding standards and especially for a team of developers (7). Six participants stated to use a linter to prevent bugs, including some specific types of errors such as security bugs, browser specific bugs and generally "*sneaky*", "*weird*" or "*dumb*" bugs. The automatic formatting of code was a motivation for six participants and three others wished to avoid typing mistakes. Three participants wanted to uphold best practices in code, where one was not very positive in regards to the language itself: "*JavaScript is a dumpster fire of a lan-*

[6]The reasons presented in Table 6.4 are shortened to fit in the table. The longer versions of the reasons are mentioned in the text and also shown in the survey PDF [154].

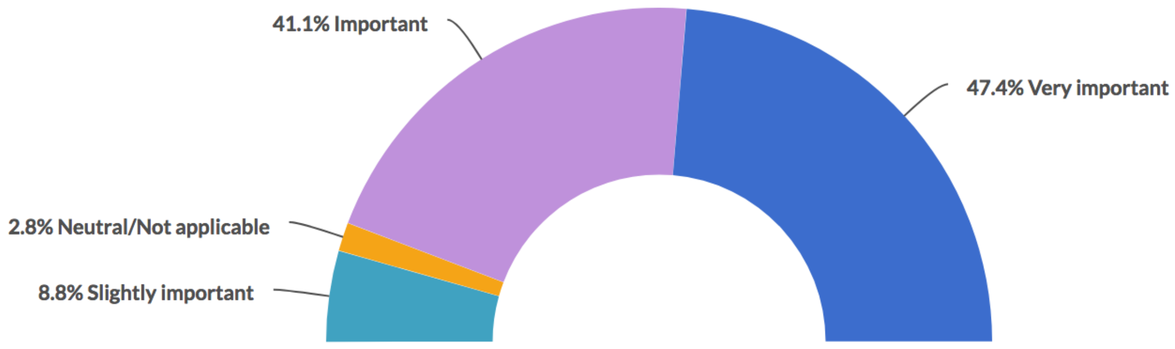
| **Reason** | **SD** | **D** | **N** | **A** | **SA** | **25th** | **Median** | **75th** |
|---|---|---|---|---|---|---|---|---|
| Catch bugs and mistakes | 1.0% | 1.6% | 4.2% | 33.7% | 59.5% | A | SA | SA |
| Avoid ambiguous code | 1.9% | 10.7% | 19.7% | 45.6% | 22.0% | N | A | A |
| Maintain code consistency | 0.6% | 1.0% | 1.6% | 27.8% | 68.9% | A | SA | SA |
| Automate code review | 3.2% | 9.1% | 23.6% | 35.6% | 28.5% | N | A | SA |
| Avoid negative comments | 16.8% | 22.0% | 34.6% | 18.1% | 8.4% | D | N | A |
| Save discussion time | 2.3% | 3.9% | 11.9% | 37.7% | 44.2% | A | A | SA |
| Learn JavaScript features | 13.9% | 27.8% | 27.5% | 23.3% | 7.4% | D | N | A |

Table 6.4: Level of agreement with reasons as to why the participants have used a linter. Columns show SD=Strongly Disagree, D=Disagree, N=Neutral, A=Agree and SA=Strongly Agree. Last three columns present the options of the first (25th%), second (median, 50%) and third (75%) quartile values.

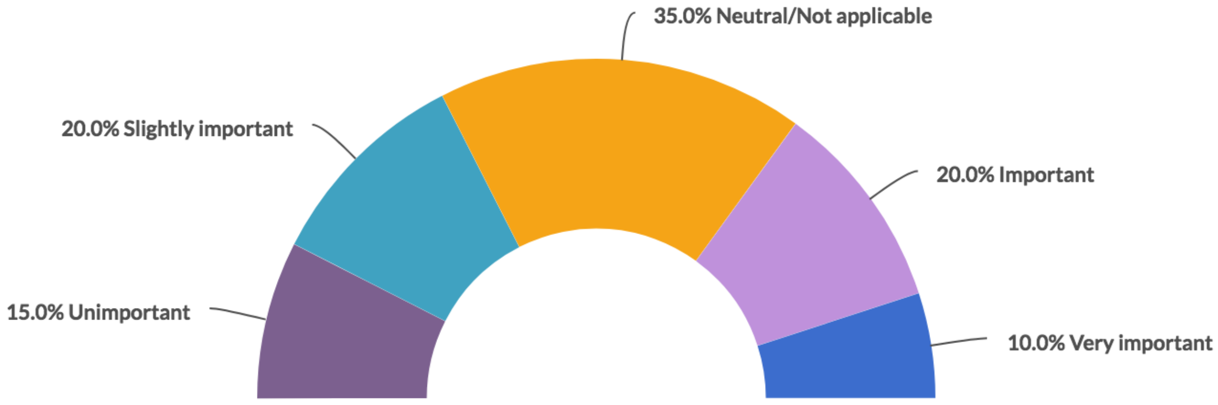
*guage and restricting it severely is the only way to get rid of the "bad parts"*.". Lastly, two participants claimed to wish to save time by using a linter. Other reasons were not shared between multiple participants (7), such as avoiding browser incompatibility, improving the code's performance and to explicitly display intentional "*odd behavior*" by using disabling comments with the linter.

There were 21 participants that had never used a linter in a JavaScript project. These participants only received two additional questions in the survey, including why they do not use a linter. Three participants explained that they simply do not need to use a linter, *e.g.*, as they already use an IDE formatter and rather rely on TypeScript for typing analysis and on team discipline and code reviews for code quality. Two participants did not find it beneficial enough to use a linter since it can complicate the build process, configuring can be demanding and it can increase the cost of the development life cycle. Two others did not have enough knowledge about linters, one was not in charge of the repository settings and three simply did not know why they had never used a linter. Other reasons shared by a single participant each were the following: there is too much effort involved in setting up a linter in a big project, it is difficult to integrate the linter with some IDEs and that other tasks have taken precedence before setting up a linter.

Lastly, all participants were asked to evaluate how important they consider it to be to use a linter in a JavaScript project. Figure 6.4 shows the participants' ranking, separately for those who had used a linter and those who had not. For those that had used a linter (Figure 6.4a), 88.8% thought it was important or very important to use one. Interestingly, not a single participant considered it to be unimportant, which is the only choice that is missing from the figure. The distribution is completely different for participants that had not used a linter before (Figure 6.4b). Most participants did not state an opinion (35.0%) while almost as many considered it to be important or very important (30.0%) as well as unimportant or slightly important (35.0%).



(a) Participants that had used a linter before (285 total)



(b) Participants that had not used a linter (20 total)

Figure 6.4: The importance of using a linter in a JavaScript project.

#### 6.3.2.2 Preferred Linter

Regarding which linters the participants had used, they had the option to select all linters they had ever used out of the options shown in Figure 6.5. ESLint was by far the most commonly used linter where 87.1% had used it, followed by JSHint with 53.9% of participants having used it. Almost half of the respondents (49.0%, 124 participants) answered an "optional" question asking whether they preferred one linter over another, and if so, why. Out of these, 75.2% stated to prefer ESLint over other linters, most commonly due to its high level of configurability and extensibility with plugins. More specifically, the following were the reasons mentioned as to why the participants prefer ESLint over other linters:

1. Very customizable with a high level of configurability (32)
2. Extensible and modular with custom plugins (25)
3. An active community, keeping the linter well documented and maintained (21)
4. Modern in how it supports new JavaScript language features, such as ES6 and JSX (15)

5. The presence of useful presets and in particular Airbnb (10)
6. Highly flexible (9)
7. Vast amount of available rules (9)
8. Many features offered (6)
9. Easy to set up and use (5)
10. Is the current *de facto* standard for JavaScript projects (5)
11. Fast execution (4)
12. Integrates well with other tools (4)
13. The presence of the auto fixing feature (4)
14. Is a single solution for roles that older linters performed separately (3)
15. Good for big projects and organizations (3)
16. Has good default settings (3)
17. Is not opinionated (2)

Some responses (20) could however not be be grouped into any of the above categories as the participants did not state any specific reasons for their choice, such as the comment: "*ESLint is pretty dope.*". In general, statements about ESLint tended to be extremely positive, such as the following comments: "*ESLint. Up to date, extendable, covers all grounds, straight up the best.*" and "*ESLint has everything I could ever ask for as far as features, extensibility, and support are concerned.*".

For JSHint, JSLint and JSCS, very few reasons were reported. One described JSHint as "*less pedant*" and another described JSCS as as very customizable. JSLint was considered to be a good tool by one participant as it is very strict and does not require configurations, while another wrote: "*JSLint, it no longer makes me cry.*". Several participants preferred to use TSLint (17) where nearly all (16) gave a very straight-forward explanation in line of the following comment: "*TSLint because, well, Typescript...*". Eight participants favored Standard where the majority explained their reason to be the fact that one does not have to configure the tool. Another six participants preferred to use Prettier (added as an extra option) but mostly without any further explanations. Lastly, two participants mentioned XO as there are no configurations needed for the tool. Additional nine responses could not be categorized with any codes as no linter was specified in the answers, in spite of the instructions to do so. Four others did not claim to prefer any linter over another.

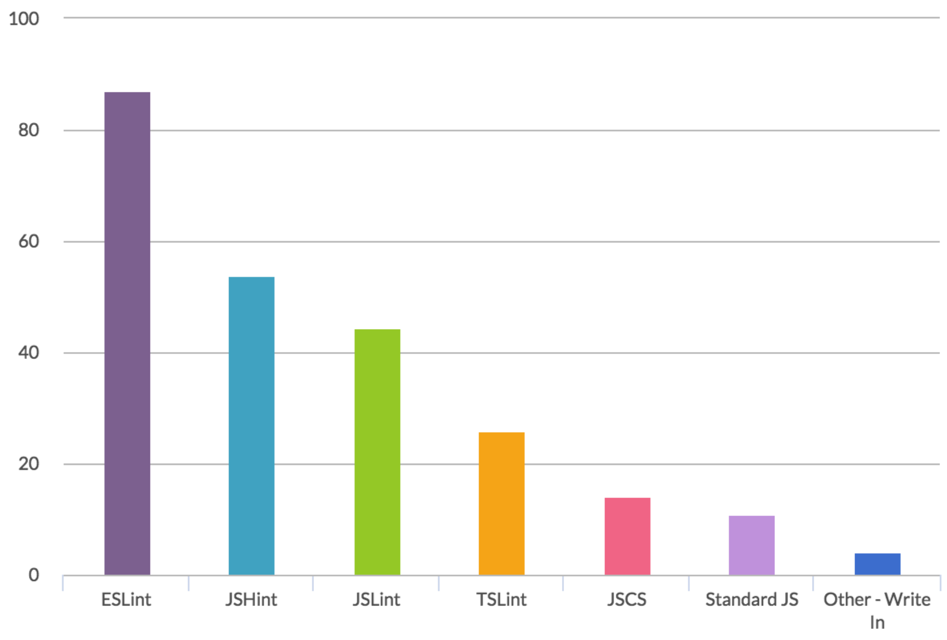


Figure 6.5: The percentage of participants that had used one of the following linters: ESLint, JSHint, JSLint, TSLint, JSCS and Standard JS.

**RQ$_6$ Why do developers use linters and what are the most important features?**
In general, developers find it to be important to use linters in JavaScript projects. They use a linter most commonly to maintain code consistency and to catch bugs and typing mistakes. Some developers do not use a linter as they rely on other methods to perform the linter's tasks or do not find it worth the effort to use one. ESLint is by far the most commonly used and liked linter, due to how customizable it is and extensible with plugins and also for its active community with well documented and up-to-date features.

### 6.3.3 RQ$_7$ Configuring Linters

We examined what methods developers use to configure linters and in particular how they use presets and then which presets they prefer to use.

Participants were asked, based on their previous experience, which methods they had used to select the rules that are included in a linter configuration file. They were presented with 11 options as shown in Table 6.5, that were derived from the interviews. Participants could select all methods they had ever used and also had the chance to add their own methods to the list. Using a preset is the most common method to use to configure a linter (70.2%), which will be further analyzed later in this section. More than half of the participants choose rules that fit the current style of a project (56.0%) or have used default configurations (51.1%). Plugins are also commonly used (41.1%), along with enabling rules that come up in discussions on a project (37.9%). Some participants do not care about which rules are enabled (10.4%) but only very few had never configured a linter themselves (1.3%).

| **Reason** | **Frequency** |
|---|---:|
| Use a preset | 70.2% |
| Choose rules that fit the current style of a project | 56.0% |
| Use default configurations from the linter | 51.1% |
| Use a plugin | 41.4% |
| Enable rules that come up in discussions on a project (e.g. on a pull request or in an issue tracker) | 37.9% |
| Choose the most commonly used style within team | 33.0% |
| Try to have the configuration as minimalistic as possible | 21.0% |
| Choose rules that involve the least effort to follow (e.g. by not having to bypass rules too often) | 19.1% |
| I don't care which rules are enabled, as long as there is a set of rules present in the project | 10.4% |
| Have the linter automatically generate configurations that fit the project | 7.4% |
| I have never configured a linter | 1.3% |

Table 6.5: Methods used to configure a linter

A few additional methods were mentioned (15) where three participants claimed to configure the linter according to their personal preferences while two others said to use different settings based on each project's characteristics. Three participants use a company style guide, where one explained that there is a special JavaScript committee at his workplace who maintain JavaScript coding standards for the whole company. Others reasons include using the strictest setting possible, choose rules that prevent bugs or having a big team discussion on which style to use.

We further examined the usage of presets where participants were asked whether they preferred to use a preset or to manually choose the rules to enable in a project. Figure 6.6 shows that most participants (74.8%) prefer to use a preset with some modifications to it by adding or removing rules. For the rest, more participants preferred to manually choose the rules (12.9%) rather than to solely use a preset (10.3%).

They were further asked which presets they like to use, where they could choose one or more of the following: Airbnb, Google, StandardJS, Prettier or default or recommended settings from the linter (*e.g.*, `eslint:recommended`). These options were obtained with the quantitative analysis in the last chapter but these were the most commonly used presets amongst the GitHub projects. The Airbnb preset is by far the most popular one where 62.1% of the participants liked to use it in their configurations. Default settings were also commonly preferred with 43.3% choosing the option. The three remaining presets were somewhat popular with 10.0 - 19.9% of participants liking them. Additional 24 options were added where five described custom presets created by them or their associations and three mentioned a modified version of the Airbnb preset. Four referenced plugins such as for React, despite the fact that the instructions for the question explicitly stated that only presets were of interest and not plugins. Other presets mentioned were Formidable [15], ctrl [14] and Microsoft's TSLint preset [58].

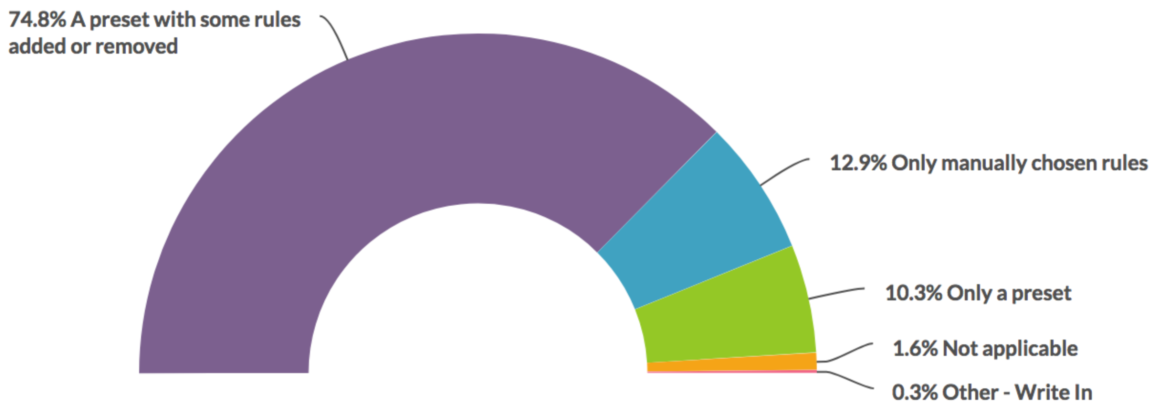


Figure 6.6: How participants use presets

**RQ$_7$ How do developers configure linters?**
Developers most commonly use presets to configure a linter, but also choose rules that fit a project's style or use default configurations from a linter. Furthermore, they generally prefer to use a preset in configurations but not on its own and rather with some rules added or removed. The preset from Airbnb is the most popular one, followed by a linters' default or recommended settings.

### 6.3.4 RQ$_8$ Important ESLint Categories and Rules

In this part we examine how participants rate the importance of the different categories from ESLint, and also for specific rules in the four main categories: Possible Errors, Best Practices, Variables and Stylistic Issues. Only participants that had used ESLint in the past received the questions that are discussed in this section, as they all relate to specific features of the tool.

More specifically, participants were asked which ESLint categories they considered important to include in configurations. Table 6.6 shows that Possible Errors is the most important category where 92.5% of the participants consider it to be either important or very important. Possible errors is then followed by ECMAScript 6, Best Practices and Variables, where over 86% of the participants consider each of them to be either important or very important. For these four categories, only four participants considered any of them to be unimportant (one for Possible Errors and three for ECMAScript 6). The three remaining categories, Strict Mode, Node.js & CommonJS and Stylistic Issues, are all considered to be important as well, but to a lesser extent than the other categories. Strict Mode is most commonly considered to be unimportant, or by 11% of the participants.

By mining projects on GitHub we saw that there are four categories that are most commonly used in configurations: Possible Errors, Best Practices, Variables and Stylistic Issues. For each of these categories we asked the participants to rate the importance of 14 rules, where the rules were chosen as described in Section 6.2.1.2. The resulting sets and

| Category | U | SI | N | I | VI | 25th | Median | 75th |
|---|---|---|---|---|---|---|---|---|
| Possible Errors | 0.4% | 1.9% | 5.3% | 33.0% | 59.5% | I | VI | VI |
| Best Practices | 0.0% | 3.0% | 8.3% | 51.1% | 37.5% | I | I | VI |
| Variables | 0.0% | 2.7% | 11.0% | 52.7% | 33.7% | I | I | VI |
| Strict Mode | 11.0% | 7.2% | 24.0% | 30.8% | 27.0% | N | I | VI |
| Node.js & CommonJS | 4.6% | 4.2% | 28.6% | 43.5% | 19.1% | N | I | I |
| Stylistic Issues | 1.5% | 7.3% | 13.0% | 52.5% | 25.7% | N | I | VI |
| ECMAScript 6 | 1.1% | 3.0% | 9.1% | 39.2% | 47.5% | I | I | VI |

Table 6.6: The importance of each ESLint category. Columns show U=Unimportant, SI=Slightly Important, N=Neutral, I=Important and VI=Very Important. Last three columns present the options of the first (25th%), second (median, 50%) and third (75%) quartile values.

their ranking are shown in Table 6.7 where the column origin describes from which list the rule was retrieved, according to the criteria explained before. The value *1/e/n* stands for the first (*1*) rule in the list of the most commonly enabled (*e*) rules for projects that do not use a preset (*n for no*). Similarly, *2/d/y* stands for the second (*2*) rule in the list of the most commonly disabled (*d*) rules for projects that do use a preset (*y for yes*). If rules were present on more than one of these sets (*e.g.*, on both top enabled rules with and without a preset), they take precedence in the following order: 1) enabled rules without a preset, 2) enabled rules with a preset, 3) disabled rules with a preset and 4) disabled rules without a preset. It is thus more common to see the label *-/e/n* in the results as it is prioritized before other origins. After each of these questions, the participants had the optional opportunity to explain their choices or to mention additional rules that they deemed particularly important (however not many used this opportunity).

The rules in the Possible Errors category are shown in Table 6.7a. Using the median rating value, eight of the 14 rules are thought to be either important or very important. The rules that originate from the list of enabled rules without using a preset are generally thought to be the more important ones, where `no-dupe-keys` is the most important rule, followed by `no-unreachable` and `no-invalid-regexp`. Interestingly, `valid-jsdoc` is one of the least important rules in this set, even though it is the most commonly enabled rule when it comes to projects that also use presets. Moreover, the rule `no-console` seems to be the most debatable rule where it is thought to be unimportant by 25.6% of participants but also (very) important by 35.2% of participants. This rule was commented on by four respondents, saying that it is sometimes not necessary to include the rule on single-person projects and that most logging should be disallowed except for logging errors. One participant mentioned the importance of the additional rule `no-await-loop` due to potential performance problems. Others claimed that the rules that cause actual bugs are generally the most important ones (without mentioning any specific rules).

The Best Practices rule set is shown in Table 6.7b. In this case, most rules are thought to be (very) important or 12 out of the 14 listed rules. The rules `eqeqeq` and `no-eval` appear to be the most important ones where 90.4% and 81.1% consider them to be either impor-

| Rule | Origin | U | SI | N | I | VI | 25th | Median | 75th |
|---|---|---|---|---|---|---|---|---|---|
| no-dupe-keys | 1/e/n | 1.6% | 3.2% | 7.1% | 37.9% | 50.2% | I | VI | VI |
| no-unreachable | 2/e/n | 2.0% | 5.1% | 7.1% | 39.4% | 46.5% | I | I | VI |
| no-irregular-whitespace | 3/e/n | 5.9% | 10.7% | 13.8% | 41.5% | 28.1% | N | I | VI |
| no-func-assign | 4/e/n | 5.2% | 11.6% | 16.7% | 37.1% | 29.5% | N | I | VI |
| no-invalid-regexp | 5/e/n | 1.2% | 5.9% | 11.9% | 43.1% | 37.9% | I | I | VI |
| valid-jsdoc | 1/e/y | 24.2% | 16.3% | 27.8% | 23.0% | 8.7% | SI | N | I |
| no-console | 2/e/y | 25.6% | 18.8% | 20.4% | 24.0% | 11.2% | U | N | I |
| no-cond-assign | 3/e/y | 10.7% | 8.7% | 20.6% | 34.4% | 25.7% | N | I | VI |
| no-inner-declarations | 4/e/y | 23.3% | 15.8% | 26.9% | 26.5% | 7.5% | SI | N | I |
| no-extra-semi | 5/e/y | 17.5% | 12.7% | 19.1% | 29.9% | 20.7% | SI | I | I |
| no-extra-parens | 2/d/y | 16.4% | 15.6% | 24.4% | 30.8% | 12.8% | SI | N | I |
| no-empty | 5/d/y | 9.1% | 11.8% | 26.0% | 37.8% | 15.4% | N | I | I |
| no-constant-condition | 6/d/y | 13.2% | 11.6% | 29.2% | 33.2% | 12.8% | N | N | I |
| no-extra-boolean-cast | 6/d/n | 11.5% | 16.6% | 31.6% | 33.6% | 6.7% | SI | N | I |

(a) Possible Errors

| Rule | Origin | U | SI | N | I | VI | 25th | Median | 75th |
|---|---|---|---|---|---|---|---|---|---|
| eqeqeq | 1/e/n | 1.6% | 2.0% | 6.0% | 32.3% | 58.1% | I | VI. | VI |
| curly | 2/e/n | 4.0% | 8.5% | 12.1% | 39.9% | 35.5% | I | I | VI |
| no-caller | 3/e/n | 10.5% | 13.4% | 28.7% | 27.1% | 20.2% | N | I | VI |
| no-eval | 4/e/n | 4.0% | 3.2% | 11.6% | 24.9% | 56.2% | I | VI | VI |
| no-with | 5/e/n | 8.1% | 11.7% | 29.1% | 21.1% | 30.0% | N | I | VI |
| no-redeclare | 6/e/n | 4.9% | 9.0% | 11.4% | 35.5% | 39.2% | N | I | VI |
| wrap-iife | 7/e/n | 8.9% | 10.6% | 23.2% | 39.4% | 17.9% | N | I | I |
| no-extend-native | 8/e/n | 10.1% | 9.7% | 20.6% | 28.7% | 30.8% | N | I | VI |
| no-unused-expressions | 5/e/y | 2.4% | 6.5% | 11.7% | 41.3% | 38.1% | I | I | VI |
| no-multi-spaces | 6/e/y | 13.7% | 12.5% | 23.0% | 31.9% | 19.0% | SI | I | I |
| no-param-reassign | 1/d/y | 10.0% | 16.9% | 18.1% | 30.9% | 24.1% | SI | I | I |
| consistent-return | 2/d/y | 12.9% | 14.5% | 20.2% | 28.2% | 24.2% | SI | I | I |
| vars-on-top | 3/d/y | 32.9% | 15.3% | 18.9% | 22.1% | 10.8% | U | N | I |
| no-warning-comments | 3/d/n | 27.6% | 16.3% | 37.8% | 12.6% | 5.7% | U | N | N |

(b) Best Practices

tant or very important. These are then followed by `curly` and `no-unused-expressions`. Furthermore, the rules `vars-on-top` and `no-warning-comments` are notably the least important ones, which were retrieved from the lists of the most commonly disabled rules. One participant shared his discontent with the rule `no-warning-comments`, saying that he did not know about it previously and that he "*would probably flip a table if it broke a CI build*". Another participant expressed that this rule should only be a part of a specific setup for deployment or production and not for development. A general observation was also shared by another participant saying that it is good to enforce some rules for production code but that

| Rule | Unimp. | SI | N | I | VI | 25th | Median | 75th |
|---|---|---|---|---|---|---|---|---|
| init-declarations | 22.6% | 14.0% | 38.7% | 15.6% | 9.1% | SI | N | N |
| no-catch-shadow | 8.6% | 9.4% | 28.7% | 33.6% | 19.7% | N | I | I |
| no-delete-var | 18.0% | 11.5% | 26.2% | 25.8% | 18.4% | SI | N | I |
| no-label-var | 7.8% | 9.5% | 28.8% | 32.5% | 21.4% | N | I | I |
| no-restricted-globals | 8.9% | 8.9% | 26.8% | 32.5% | 22.8% | N | I | I |
| no-shadow | 8.6% | 10.2% | 23.0% | 32.4% | 25.8% | N | I | VI |
| no-shadow-restricted-names | 4.1% | 7.8% | 27.9% | 32.4% | 27.9% | N | I | VI |
| no-undef | 3.2% | 4.4% | 10.4% | 28.5% | 53.4% | I | VI | VI |
| no-undef-init | 20.4% | 15.9% | 25.7% | 22.9% | 15.1% | SI | N | I |
| no-undefined | 14.3% | 9.0% | 17.2% | 27.0% | 32.4% | N | I | VI |
| no-unused-vars | 0.4% | 6.0% | 6.0% | 35.9% | 51.6% | I | VI | VI |
| no-use-before-define | 2.8% | 5.7% | 12.1% | 30.4% | 49.0% | I | VI | VI |

(c) Variables. Origin is not included as there are only 12 rules in this category and thus no need to use any criteria to choose a subset of the rules.

| Rule | Origin | U | SI | N | I | VI | 25th | Median | 75th |
|---|---|---|---|---|---|---|---|---|---|
| quotes | 1/e/n | 5.8% | 9.5% | 3.7% | 40.3% | 40.7% | I | I | VI |
| semi | 2/e/n | 5.8% | 4.5% | 14.9% | 30.6% | 44.2% | I | I | VI |
| indent | 3/e/n | 2.9% | 5.4% | 5.4% | 29.5% | 56.8% | I | VI | VI |
| brace-style | 4/e/n | 3.7% | 4.6% | 12.9% | 42.3% | 36.5% | I | I | VI |
| comma-style | 5/e/n | 6.3% | 8.8% | 10.1% | 43.7% | 31.1% | N | I | VI |
| comma-dangle | 6/e/n | 7.5% | 10.9% | 15.1% | 37.7% | 28.9% | N | I | VI |
| space-before-blocks | 7/e/n | 7.9% | 11.3% | 15.8% | 39.6% | 25.4% | N | I | VI |
| eol-last | 8/e/n | 15.0% | 13.3% | 18.8% | 30.4% | 22.5% | SI | I | I |
| linebreak-style | 5/e/y | 8.3% | 6.7% | 10.4% | 34.2% | 40.4% | N | I | VI |
| space-before-function-paren | 6/e/y | 11.7% | 11.7% | 15.5% | 36.4% | 24.7% | N | I | I |
| no-underscore-dangle | 1/d/y | 29.9% | 14.1% | 28.2% | 14.9% | 12.9% | U | N | I |
| func-names | 3/d/y | 16.3% | 15.8% | 29.6% | 24.6% | 13.8% | SI | N | I |
| max-len | 4/d/y | 14.9% | 22.8% | 18.7% | 26.1% | 17.4% | SI | N | I |
| no-ternary | 3/d/n | 63.8% | 14.6% | 12.1% | 7.1% | 2.5% | U | U | SI |

(d) Stylistic Issues

Table 6.7: The importance of a subset of rules in the categories Possible Errors, Best Practices, Variables and Stylistic Issues. Columns show U=Unimportant, SI=Slightly Important, N=Neutral, I=Important and VI=Very Important. Last three columns present the options of the first (25th%), second (median, 50%) and third (75%) quartile values.

they "*should not hinder one's development style*". The less rated rule `vars-on-top` was also said to be unimportant as with ES6 it is possible to have block scope local variables with `let` and `const` (instead of using the `var` statement which defines a variable regardless of block scope). Lastly, one respondent discussed the rule `no-extend-native` and the two opposing views about extending native objects. Being in favor of the feature, he described it as one of the advantages of JavaScript where avoiding it is "*akin to leaving the protective plastic on a couch*".

For the Variables category in Table 6.7c the set simply consists of all available rules as there are only 12 in the category. Nine out of the 12 rules are considered to be (very) important by the participants. Three rules seem to be particularly important, `no-unused-vars`, `no-undef` and `no-use-before-define`, where 87.5%, 81.9% and 79.4% of the participants, respectively, consider them to be either important or very important. No specific comments were made about any of these rules.

The subset of rules for the Stylistic Issues category is shown in Table 6.7d. 10 out of the 14 rules are considered to be (very) important, but those are exactly the rules that were retrieved from the lists of the most commonly enabled rules. One rule is considered to be very important (based on the median value), `indent`, where 86.3% consider it to be either important or very important. There are three other rules which over 40% of the participants consider to be very important: `quotes`, `semi` and `linebreak-style`. On the contrary, the rule `no-ternary` is unquestionably the least important rule where 63.8% of participants consider it to be unimportant and only 9.6% consider it to be (very) important. The rule `no-underscore-dangle` is also relatively unpopular amongst the participants as 29.9% deem it unimportant. Lastly, the rule `comma-dangle` was mentioned by two participants to be particularly good to catch errors, where it was described as "*unquestionably the source of most of my lint errors*"[7]. Other participants generally described rules from this category to be beneficial for code readability within a team and for simple code merging.

**RQ$_8$ Which types of rules are important to use for a linter?**
Possible Errors is the most important category to include in configurations, where Best Practices, Variables and ECMAScript 6 are also deemed important. The most important rules for each of the four main categories are the following:
*Possible Errors*: `no-dupe-keys`, `no-unreachable` and `no-invalid-regexp`.
*Best Practices*: `eqeqeq` and `no-eval`.
*Variables*: `no-unused-vars`, `no-undef` and `no-use-before-define`.
*Stylistic Issues*: `indent`, `quotes` and `semi`.

### 6.3.5 RQ$_9$ Challenges of Using a Linter

When interviewing the developers we saw that they faced several challenges when using a linter. Here we examine those challenges and further see whether the participants have experienced false positives while using a linter.

The participants were asked which challenges they had faced in their previous experience with using a linter. The options were mostly derived from the interviews and are shown in Table 6.8. One additional option was provided, namely: *too many warnings/errors outputted from the linter*, as it was suggested by one of the pilot test participants and has been

[7]The rule `comma-dangle` was previously a part of the Possible Errors category due to the known errors these dangling commas could cause in Internet Explorer 8 (and older IE versions). The rule was however moved to the Stylistic Issues category when older versions of Internet Explorer stopped being supported by Microsoft, thus making this error less relevant.

reported in related research to be a common problem with using ASATs [108, 98]. Almost half of the participants (47.0%) reported to have problems with agreeing on which rules to use within a team. Another 38.3% agreed that creating or maintaining configurations was a challenging part of using a linter. Around a third of the participants agreed with the rest of the reasons, expect for the lack of dynamic analysis which was the least commonly reported challenge, but still with a quarter (24.5%) of the respondents claiming it to be a problem. Only 8.4% had not experienced any challenges with using a linter. Participants also had the opportunity to add other challenges which 15 of them did. Amongst these were three challenges mentioned by more than one participant:

1. Getting team members to use a linter (3)
2. Integration with IDE or other tools (3)
3. When ESLint, presets or plugins change (2)

This last challenge was described by one participant as: "*I detest it if the rules in a preset change*" but this was also mentioned by some of the interview participants. Other reported challenges included difficulty with writing new linting rules and the fact that using the tool during development is "*obnoxious*" but is more beneficial when it comes to the time of committing code.

| **Challenge** | **Frequency** |
|---|---:|
| Agreeing on which rules to use within a team | 47.0% |
| Creating or maintaining configurations | 38.3% |
| The presence of false positives | 33.9% |
| Enforcing rules in a team | 33.6% |
| Enabling rules in an existing project | 33.2% |
| Too many warnings/errors outputted from the linter | 29.2% |
| The lack of analysis for dynamic features of JavaScript | 24.5% |
| No challenges | 8.4% |

Table 6.8: The challenges that participants face when using a linter

A third of the participants claimed that the presence of false positives is indeed a challenging part of using a linter. Participants were further asked about this topic where for each linter they had used (as answered in one of the earlier questions) they reported how frequently they had experienced false positives. The participants' experience with false positives is shown in Table 6.9. According to the median value, false positives are either rare or very rare with all the linters except for JSLint. The median value for JSLint is N/A (not applicable), which was intended as an option when the participants can not evaluate their experience with false positives for a certain tool, *e.g.*, because too much time has passed since they used the linter and perhaps do not remember these details. For JSLint, we therefore interpret that the median experienced frequency lies somewhere between being rare and occasional. Generally, very few participants claimed any linter to have frequent false positives, where only with JSLint and TSLint the percentage went over 10% (12.2% and

10.4%, respectively). The results are mostly similar for ESLint and StandardJS, which is expected as StandardJS uses ESLint to function.

| **Linter** | **N** | **VR** | **R** | **N/A** | **O** | **F** | **VF** | **25th** | **Median** | **75th** |
|---|---|---|---|---|---|---|---|---|---|---|
| ESLint | 22.8% | 27.8% | 22.4% | 3.1% | 19.3% | 2.7% | 1.9% | VR | VR | N/A |
| JSHint | 14.3% | 18.8% | 16.9% | 20.1% | 24.0% | 5.2% | 0.6% | VR | R | O |
| JSLint | 11.5% | 18.3% | 19.1% | 17.6% | 21.4% | 7.6% | 4.6% | VR | N/A | O |
| TSLint | 20.8% | 29.9% | 13.0% | 5.2% | 20.8% | 5.2% | 5.2% | VR | VR | N/A |
| JSCS | 9.8% | 24.4% | 17.1% | 19.5% | 26.8% | 2.4% | 0.0% | VR | R | O |
| StandardJS | 25.8% | 22.6% | 16.1% | 12.9% | 19.4% | 3.2% | 0.0% | N | R | N/A |

Table 6.9: The frequency of experienced false positives when using a linter, ordered by the most commonly used linters. Columns show N=Never, VR=Very Rare, R=Rare, N/A=Not Applicable, O=Occasionally, F=Frequently and VF=Very Frequently.

**RQ$_9$ What is challenging about using a linter?**
The most commonly faced challenges are to agree on which rules to use within a team and to create or maintain configurations. It is generally rare to experience false positives with most JavaScript linters.

## 6.4 Discussion

### 6.4.1 Participants

Regarding the distribution of the survey, it seems as if Twitter is not the ideal place to promote this type of questionnaire. Even though it was exposed to a large quantity of people, it did not attract many responses. It was much more effective to promote the survey in smaller communities that are known to include members of the target population. Interestingly, the participants directed from the EchoJS community had a higher average experience than those from the other sources, suggesting it to be a good source for finding experienced JavaScript developers.

### 6.4.2 Why Developers Use Linters

ESLint is by far the most popular JavaScript linter out of those that are available today. The main reasons for its popularity are the various features of the tool, especially in terms of how customizable it is and how extensible it is with plugins. ESLint is therefore much more flexible than other available linters, allowing developers to use it exactly as they wish. Somewhat contrary to this, developers also appreciate the fact that they can use presets such as Airbnb (and use it frequently), which in theory removes some needs for configurability by not having to configure all rules that are used. However, developers still tend to add or remove some rules in addition to using a preset, proving it to be important to at least being able to tweak the settings that are used so they fit better to a project. Another big factor to

ESLint's popularity is the active community around the tool and how well it is maintained and supported. This goes to show how important it is to have tools well documented and frequently updated. It is also important that they stay up to date with new features and trends in the language that they support.

There appear to be several reasons for developers to use a linter in general, but most importantly to maintain code consistency and to prevent possible errors in code. ESLint provides rules to perform both of these tasks, whereas other linters such as JSCS and JSHint rather focus on one role each. It is therefore also important that these tools have a wide variety of functionality.

### 6.4.3 Configuring Linters

Just over half, or 51.1%, of the participants had ever used default configurations, which is a relatively low percentage compared to common usage patterns for ASATs [69]. This might be due to the fact that the majority of the survey's participants use ESLint which does not have any default configurations since version 1.0.0 (July 2015). The most popular method however to configure a linter is to use a preset or to choose rules that fit the style of the project. The usage of presets is thus now a well established feature of the linter where people heavily rely on guidance from others in choosing which rules to use. Only 12.9% of the participants preferred to only manually choose the rules and not use any presets. However, preset-users also prefer to make some modifications to the presets they use, exploiting ESLint's range of customizability.

Airbnb is by far the most popular preset amongst the participants. As touched upon before, the preset includes rules from various categories of ESLint, thus providing an overall setting to use in a project, *e.g.*, both to maintain a consistent code style and to catch errors. This wide range of settings might therefore be the reason for its popularity. The second most popular preset is the default one of the linter, which we made to include ESLint's recommended settings. As this preset mostly contains rules that relate to possible errors, this might be a popular choice for those that do not want a broad setting but instead one that focuses on discovering possible bugs in the code.

### 6.4.4 Important ESLint Categories and Rules

According to the participants of this survey, Possible Errors is the most important category to include in a project's configuration, where however all other categories were also deemed to be important. It is interesting that the most commonly reported reason for using a linter was to maintain code consistency, but still Possible Errors is deemed far more important than Stylistic Issues and also Best Practices (which could both be considered appropriate for the goal of upholding code consistency). Even when looking only at the responses from those that strongly agreed with code consistency as a reason for using a linter, Possible Errors is still rated more important than these two categories. Catching bugs and typing mistakes was however the second most common reason for using a linter, which can correspond to the Possible Errors category. This is indeed the same pattern as was discovered when interviewing developers in Chapter 4: maintaining a consistent code style is the most common reason for using a linter but for most it is still more important for a linter to detect errors.

Some rules were considered to be more important than others, where the highest rated ones also tended to be commonly enabled in the previously analyzed GitHub projects, thus complementing each other. From the categories Possible Errors, Best Practices and Variables, the most important rules are all known to relate to possible bugs which can then be prevented by enabling said rules. For the last category, Stylistic Issues, the most important rules all have do to with the use of consistent punctuation in code.

A possible concern with this importance rating is regarding the unknown effects of the middle option labeled with *Neutral/Not applicable*. Some might have chosen the option because they did not recognize the rule (and might not have read the documentation that followed) or they might just not have a very strong opinion on the particular rule. It was more common with the Variables category that participants chose the neutral option than for any other category and there were generally fewer neutral votes for the rules that ended up being more important than others. It would be possible to conduct the analysis with all neutral votes excluded, but that was thought to alter the data too much. Regardless, it was necessary to include this option in the scale to not force any opinions on the participants.

### 6.4.5 Challenges of Using a Linter

It seems that it is generally challenging to some extent to use a linter, as only 8.4% of the participants reported that they had not faced any challenges with using such a tool. The most commonly reported challenge in using a linter was to agree on which rules to use within a team. This challenge was discovered when performing the interviews where some interviewees reported this to be the case when working with a team in an industrial setting, as opposed to working in OSS development. A high majority of the survey's participants had indeed worked on commercial software (87.0%) and fewer had developed in the OSS community (50.6%). It was not included in the wording of the question that this should only apply to an industrial setting, but with more participants having rather worked in industry, it is likely the case that it applies in that particular setting.

Even though 33.9% of the participants consider false positives to be a challenging part of using a JavaScript linter, the participants generally do not experience false positives frequently with any of the discussed linters. For all reported linters, except for JSLint, the participants' experience on the frequency of false positives is most commonly either rare or very rare. JSLint is the oldest JavaScript linter and is still maintained today, but however does not have an active community around it[8]. As discussed earlier in Chapter 4, the experienced frequency of false positives with ESLint does not compare with that of findings in literature on ASATs in general. We see now that this seems to be the general case within the JavaScript community and with most available linters.

[8]This assumptions is based on the fact that 1) Douglas Crockford is the repository's only maintainer on GitHub [40] and 2) JSLint's community on Google+ [39] has only 794 members and very few recent posts.

### 6.4.6 Limitations

It is vital for a survey that the respondents represent the target population of the study. Earlier in Section 6.2.5.3 we discussed the representativeness of the response set, which we can further evaluate based on the participants' demographical information. As discussed before, it was assumed that the sample would indeed be representative of the population of JavaScript developers as the survey was mostly only promoted in communities specifically intended for this said target group. However when distributing a survey online, one can never be sure who actually answers the questionnaire as anyone can have access to it. It turns out that the sample is in fact full of JavaScript developers where it is the main language of 96.1% of the participants, with an average experience of 5.8 years with using JavaScript and where every single respondent had used the language in the last 12 months. Furthermore, 87.0% regularly work in industry and 50.6% with OSS, which allows this analysis to cover both the commercial and open source sectors. The sample is therefore indeed representative of the population, where the members both have a sufficient and a diverse amount of experience with the language. Additionally, the sample is very diverse regarding the participants' country of residence, leading to the results not representing only one market or one part of the world. However, there is almost no gender diversity in the sample, which is in fact a common problem when performing surveys within the software engineering field [117].

Some participants had never used a linter, but only 6.3% (21) of the survey's respondents, thus not providing a great deal of feedback regarding why. Even though more feedback on this topic would have been interesting and beneficial, the survey was indeed directed towards those that had used a linter before as most of the questions were about its usage. As the survey needs to be concise and cohesive for respondents to complete it, the sacrifice was made to mostly emphasize on why and how the tool is actually used, and not on why it is not used. Furthermore, many developers out there most likely do not use the tool as they do not know much about it, but these people were less likely to take the survey as they did not recognize well its topic. The results regarding why developers do not use a linter are therefore much less reliable than those regarding why developers do use a linter.

For developers who had used a linter before, only 8.8% considered it to be only slightly important to use a linter and not a single participant thought it was unimportant. As the survey was distributed in online communities and not directly to any specific people, it is likely that those that were previously interested in the topic took the time to answer the survey. This is indeed a limitation of this distribution method, where the results might be somewhat different if more participants were less interested in the topic. However, it is likely that people who are less interested in the topic also know less about it, and thus not able to provide as accurate information as we seek after in this survey.

When asking about the presence of false positives, there is a possible concern with how the participants interpret the term *false positive*. It was discovered in the pilot tests that some participants were not sure whether the term referred to a wrongly indicated warning (as intended here) or simply a warning that a developer might not find to be appropriate in some context and might have intended on breaking a rule in that specific case. This latter condition is indeed something that the interviewees experienced where in some cases they did not think that a rule should apply, even though the tool was technically correct to report it. To try to prevent this misinterpretation, the following explanation was included with each mention of false positives in the survey: "*A false positive is a wrongly indicated error/warning. If you intend to break a rule in a specific case, it is not considered as a false positive*".

## 6.5 Conclusion

We surveyed 337 developers where we asked them about their experiences with using a linter for JavaScript projects. The questionnaire was carefully designed and pilot tested before being promoted on Twitter and in three online JavaScript community groups. It is clear that developers primarily use linters to maintain code consistency and to catch bugs and typing mistakes in code. They most commonly use ESLint out of all linters due to the many features of the tool (*e.g.*, configurable and pluggable) and its active community. Using a preset appears to be the preferred way of configuring a linter with the possibility of modifying specific rules. It is however challenging to maintain configurations for a team and to agree on which rules should be used.

# Chapter 7

# Conclusion

In this thesis we have examined why and how developers use linters for JavaScript, along with the challenges that they face. We applied three research methods to collect data where we interviewed 15 developers, analyzed over 9,500 configuration files and finally surveyed more than 300 JavaScript developers. Before discussing the combined results of the whole study, we first revisit the individual research questions for each of the three main chapters. Thereafter the results are demonstrated in a broader context where the three chapters are discussed together with the implications that they offer for developers, tool makers and researchers. Lastly, we summarize the main topics of this thesis and conclude with some final words.

## 7.1 Revisiting the Research Questions

We first revisit all three chapters and their individual research questions.

### 7.1.1 Developers' Perceptions of Linters

This study began by interviewing 15 JavaScript linter experts to understand why and how developers use linters for this notorious language. An abundance of information and experiences was collected where we obtained valuable insights into how these tools are used and which challenges developers face.

**$RQ_1$ Why do developers use linters?**
The most common reasons for using a linter are to maintain code consistency and to prevent errors. Other reasons for using a linter are to augment test suites, avoid ambiguous and complex code, perform faster code reviews, spare developers' feelings during code reviews, save time on discussing code styles and to learn about the JavaScript language.

**$RQ_2$ How are rules selected for a project?**
The methods that developers use to configure ESLint are to use an existing preset, choose rules that fit the project's style, have the linter automatically generate a set of rules, enable rules that come up in discussions during code review, have minimalistic configurations,

choose rules that involve the least effort to follow and to go with the most common style within the development team.

**RQ$_3$ What are the challenges in using a linter?**
The main challenges that developers face are to create and maintain configurations, enable rules in an existing codebase, agree on which rules to use in an industrial setting and to enforce developers to follow rules without breaking the build. Developers are generally not concerned with the lack of dynamic analysis and generally do not experience false positives.

### 7.1.2 Exploring Linter Configurations in JavaScript projects

This chapter consisted of an elaborate quantitative analysis on the ESLint configuration files of 9,548 projects on GitHub. We analyzed the prevalence of these configurations were we looked into the usage of both presets and individual rules that are specified. We further examined which types of rules are most commonly used in JavaScript projects.

**RQ$_{4A}$ How are presets and plugins used?**
Presets are used by 67% of all ESLint projects where the most popular ones are the recommended settings from ESLint and a style guide maintained by Airbnb, which account for over half of all presets that are used. Plugins are used by 27% of all projects where a plugin for React is by far the most popular one.

**RQ$_{4B}$ How frequently are rules specified?**
Projects that do not use presets specify a high number of 58 rules on average. Those that do use presets specify 10 rules on average where 70% of these projects include at least one rule. For all projects it is most common to enable rules as errors and rare to set rules as warnings.

**RQ$_{5A}$ Which are the most commonly used categories from ESLint?**
The category Stylistic Issues is enabled by the highest number of projects, followed by Best Practices and Variables, where almost half of all projects that use a preset enable at least one stylistic rule. Stylistic Issues is also the most commonly disabled category, followed by Best Practices and Possible Errors.

**RQ$_{5B}$ Which are the most commonly used rules from ESLint?**
The most commonly enabled rules are for code formatting and punctuation, *i.e.* to uphold a consistent use of quotes, semicolons and indentation. More stylistic rules are commonly enabled for projects that use presets. The most commonly disabled rules are for disallowing the use of logging to the console and the use of dangling underscores in variable names. A rule that disallows the use of the ternary operator is almost never enabled.

### 7.1.3 The Experiences and Perceptions of the JavaScript Community

For this chapter we surveyed 337 JavaScript developers where we asked various questions about the usage of linters. The results from the previous two chapters were used as input to the questionnaire where we could generalize the findings from the interviews and further examine the results from the repository mining.

**$RQ_6$ Why do developers use linters and what are the most important features?**
In general, developers find it to be important to use linters in JavaScript projects. They use a linter most commonly to maintain code consistency and to catch bugs and typing mistakes. Some developers do not use a linter as they rely on other methods to perform the linter's tasks or do not find it worth the effort to use one. ESLint is by far the most commonly used and liked linter, due to how customizable it is and extensible with plugins and also for its active community with well documented and up-to-date features.

**$RQ_7$ How do developers configure linters?**
Developers most commonly use presets to configure a linter, but also choose rules that fit a project's style or use default configurations from a linter. Furthermore, they generally prefer to use a preset in configurations but not on its own and rather with some rules added or removed. The preset from Airbnb is the most popular one, followed by a linters' default or recommended settings.

**$RQ_8$ Which types of rules are important to use for a linter?**
Possible Errors is the most important category to include in configurations, where Best Practices, Variables and ECMAScript 6 are also deemed important. The most important rules for each of the four main categories are the following:
*Possible Errors*: `no-dupe-keys`, `no-unreachable` and `no-invalid-regexp`.
*Best Practices*: `eqeqeq` and `no-eval`.
*Variables*: `no-unused-vars`, `no-undef` and `no-use-before-define`.
*Stylistic Issues*: `indent`, `quotes` and `semi`.

**$RQ_9$ What is challenging about using a linter?**
The most commonly faced challenges are to agree on which rules to use within a team and to create or maintain configurations. It is generally rare to experience false positives with most JavaScript linters.

## 7.2 Implications

In the following we combine the results from all three chapters to discuss the implications that they have for developers, ASAT makers and researchers.

### 7.2.1 Developers

The results provide motivation for developers to use a linter and also offer advice to make the usage of linters most beneficial.

#### 7.2.1.1 Motivation to Use a Linter

We have discussed several reasons as to why developers use a linter where the compilation of all these reasons serves as motivation for other developers to use a linter where they are exposed to the various benefits that the tool can provide. We have established that using a linter: a) on top of tests detects more bugs, b) helps avoid ambiguous code, c) helps maintain consistent code, d) makes code review faster and easier, e) spares newcomers' feelings when making their first contribution, f) saves time that goes into discussing code styles and g) helps developers in joining projects and to learn new language syntax and features. The findings by Beller *et al.* [69] also suggest that developers need to be made aware of the benefits of using ASATs.

#### 7.2.1.2 Rules to Choose for Configurations

There are several rules that have been identified by both experts and the JavaScript community to be particularly important to include in configurations. The same rules and others have also been proven to be in common use by the OSS community on GitHub. Considering these three different sources, we conjecture that some of these rules are indeed important and are sensible to include in configurations for other projects as well. These rules are shown in Table 7.1, in no particular order except being grouped by category. Thus when developers find themselves in the position of having to create a configuration file, they should first look to these rules as they are likely more important to include than others. This list also serves as a good starting point for teams that face the challenge of agreeing on which rules to use, where these rules have been shown to be important to many other developers. Similarly, any developer that currently maintains ESLint configurations should reevaluate them if some of these rules are missing. As we did not discuss with or ask any of the participants about the rules in the categories ECMAScript 6, Node.js & CommonJS and Strict Mode, we do not make any assumptions or recommendations regarding those.

#### 7.2.1.3 Methods to Configure Linters and Enable Rules

As creating and maintaining configurations can be challenging, developers can find several ways to choose and prioritize rules. We present different methods to choose the appropriate rules for a project, such as using presets, fitting the rules to the existing style or using discussions in pull requests as input. We further report on the presets that developers like to use, *e.g.*, Airbnb, which can give other developers an incentive to use those as well as they receive a wide endorsement from the JavaScript community.

Moreover, we recommend developers to include a linter from the beginning of a project as enabling a linter at a later stage can involve substantial effort and risk. Additionally,

| **Rule** | **Category** |
|---|---|
| no-dupe-keys | Possible Errors |
| no-unreachable | Possible Errors |
| no-invalid-regexp | Possible Errors |
| no-irregular-whitespace | Possible Errors |
| no-unused-vars | Variables |
| no-undef | Variables |
| no-shadow-restricted-names | Variables |
| eqeqeq | Best Practices |
| no-eval | Best Practices |
| curly | Best Practices |
| no-caller | Best Practices |
| semi | Stylistic Issues |
| indent | Stylistic Issues |
| quotes | Stylistic Issues |
| comma-dangle | Stylistic Issues |

Table 7.1: Most important rules to include in configurations

Ayewah *et al.* [64] also found that FindBugs users are less likely to fix warnings that apply to older code than to new code.

In existing projects, developers should enable rules carefully and incrementally. This should be done to minimize the risk of introducing bugs and the risk of creating frustration within a development team. Furthermore, by incrementally introducing rules and continuously fixing the corresponding warnings, the number of warnings can be kept to a minimum, making developers more likely to examine and fix them. Previous research has also reported that many ASATs output too many warnings which makes it more difficult to use these tools [108, 77, 139].

### 7.2.2 Tool and Preset Makers

The results help ASAT creators to improve future tool editions by understanding the needs and challenges of developers when using a linter.

#### 7.2.2.1 Languages to Focus On

Linters can be especially useful for dynamic languages as errors can be easily introduced because of the dynamic typing in JavaScript. Tool creators should therefore focus their attention on dynamic languages. Furthermore, as JavaScript code can be written so freely and in many different ways, it can be even more valuable to maintain code consistency. Beller *et al.* [69] also found that dynamically typed languages seem to benefit more than static languages from the usage of ASATs.

#### 7.2.2.2 Topics to Focus On

We have seen the main reasons as to why developers choose to use linters along with which categories they find to be the most important ones. Tool creators should therefore place emphasis on these topics that developers are most concerned with. Both the interviewees' and survey respondents' main reasons for using linters were to maintain code consistency and to catch possible bugs in code. While stylistic rules are used the most often in configurations, developers claim that bug-catching rules are far more important. These two topics should therefore be of main interest for tool makers when deciding on what type of functionality to implement.

#### 7.2.2.3 Rules to Focus On

Table 7.1 displays the rules that are deemed the most important based on this study's results. This list of rules can be of guidance to both preset makers and tool makers. As these rules seem to be the most important ones, preset makers should make sure to enable them in their configurations. The creators of ESLint should also consider to include them in their recommended settings (nine out of the 15 rules are not yet included).

This list of rules could be further expanded and ordered based on the rules' importance, which could be used by ESLint to assign a priority to all available rules. It has been shown in this study that it can be difficult to configure a linter, especially when one needs to go through all available rules to choose from. Partly for that reason, the majority of developers choose to use a preset in their configurations, while also making some modifications to their settings. The process of going through all 236 ESLint rules could be made easier by assigning them with an importance rating which could be based on this study's results, *i.e.* on the importance reported by the JavaScript community and the frequency of the rules being enabled in GitHub projects.

On a similar note, we can also identify some rules that are rarely used or commonly disabled and are not thought to be important by the JavaScript community. These rules include `no-ternary` (Stylistic Issues), `no-underscore-dangle` (Stylistic Issues) and `vars-on-top` (Best Practices). Preset creators should make sure to not enable these rules as most developers do not want them to be used. Perhaps ESLint should even consider removing these unpopular rules, as while the great amount of rules is one of the tool's best qualities, it can also make the process of configuring the tool all the more confusing.

In general, preset makers should be careful to not update the rule selection too often. Developers that use the presets can get frustrated when they have to do frequent changes to their code due to the presets being updated.

#### 7.2.2.4 Features to Focus On

We have reported several reasons why developers prefer to use one specific linter over another, where they most commonly prefer using ESLint. These reasons describe the features of the tool that are important to the users, such as for the tool to be configurable, pluggable, well supported and up-to-date with new language features. Creators of other ASATs should look to these reasons to see what makes a tool well appreciated by the public, to then include in their own tools as well. For the maintainers of ESLint, they should continue to focus on having these features available and well implemented.

It is very common to use a preset in configurations for ESLint, making it an established part of configuring a linter. Other linting tools and general ASATs should therefore consider offering this method for configuring these tools. Most developers also manually specify or change some rules, again showing how important it is to be able to customize linters. This is especially relevant for rules that relate to stylistic issues where developers have diverse opinions on code styles. It should therefore be an industry standard to be able to have custom configurations for an ASAT.

The creators of ESLint should also keep track of the popularity of commonly used plugins and consider whether they should be included as default rules to be provided by the tool. ESLint already contains rules for Node.js which are specific to a particular developing environment for JavaScript. It is apparent that the plugin for React is by far the most popular one, providing ESLint with a reason to create a new category for these rules. Doing so would simplify dependencies for ESLint users as this plugin would already be included in the tool.

Another feature that would simplify the development process would be to have a built-in feature which enables developers to have separate configurations for development and production. The configurations that are intended for production code could then be applied when pushing code to a specific branch in a project's repository and be stricter than the configurations for work-in-process code. As an example, developers could then log freely with the console during development and debugging but be sure that no log statements would slip into production code.

Furthermore, as it can be difficult to enforce linting rules without making them break the build, other methods would be beneficial to encourage developers to fix warnings. Sadowski *et al.* [141] had a similar experience when introducing ASATs at Google, where developers would ignore warnings that did not cause the build to fail. Tool creators should therefore find and employ other ways to make developers feel responsible to fix them. For example, this could be done by linking the output of the tool to the project repository, to make a somewhat more personal connection to the developers that are responsible. A warning could perhaps be marked with the name of the developer who introduced it, which could motivate him or her to fix the warning.

Lastly, tool makers should not put excessive effort into dynamic analysis for JavaScript. Most of the participants in this study were not bothered with the fact that linters are not able to handle the dynamic features of JavaScript and do not see it as a challenge with using these tools.

### 7.2.3 Researchers

The results offer ample opportunities for further research into these findings. Example research directions are listed below.

#### 7.2.3.1 The Effects of Using Linters During Code Review

Balachandran [68] conducted a study where ASATs were applied on code changes that were subject to code review, and asked developers whether they agreed with the warnings reported by the tool. Similarly, Panichella *et al.* [131] studied whether warnings were being removed in code reviews, both in projects that use ASATs and for some that do not. Both results indicated that code reviews would benefit from using ASATs by automating some of the work that developers perform, which is what was also claimed by our study participants. More research needs to be conducted to understand the effects of linters during review of JavaScript code to further evaluate how beneficial it is to apply them. Results could also show which warnings developers more often remove from source code, which could provide us with further insight into which rules are important for developers.

#### 7.2.3.2 Onboarding Newcomers

Steinmacher *et al.* [147] studied the barriers that newcomers face when making their first contribution in an OSS project. Social barriers that were discovered include late responses and negative or even impolite feedback. Technical barriers include bad code quality, code complexity and lack of code standards. Research should be conducted to see whether newcomers face less of these barriers in projects that use a linter, as the linter performs part of the code review with a non subjective feedback and also helps with maintaining better code quality.

#### 7.2.3.3 Differences in Configurations

Beller *et al.* [69] studied in large scale the configurations of ASATs and how those configurations change over time. Since several participants claim that it is difficult to enable linters for existing code, it would be intriguing to further research how configuration files in projects that enable a linter early on differ from those in projects that enable a linter at a later stage. Furthermore, many participants explained that they try to create the configuration so that it fits the project in the best way possible. It would be valuable to know how thoroughly projects follow these configurations, providing insight into how well the configuration reflects the project style and how much developers care about upholding these code standards.

#### 7.2.3.4 Custom Rules

We reported that JavaScript developers find it important for linters to be extensible which makes it possible to enable custom rules and for anyone to create them. However, Beller *et al.* [69] showed that custom rules are rarely applied, accounting for only 5% of specified rules in configurations. Furthermore, Christakis and Bird [74] found that most developers do not use or do not care about this functionality. It would be intriguing to study if, and then why, JavaScript developers write and use more custom rules than for ASAT users of other programming languages.

#### 7.2.3.5 False Positives

Developers very rarely experience false positives when using ESLint, and also rarely when using other known linters. These findings are opposite to what previous literature reports on general ASAT usage [164, 108, 71, 100]. As previously discussed, this might be due to the relatively simplistic analysis methods that are applied in linters and the non-complex issues they identify. However, we have also seen that an important factor for choosing ESLint as a linting tool is the customizability it offers. This feature is indeed used as all projects apply configurations to some extent, resulting in more relevant rules being reported for a project. The fact that developers tend to categorize irrelevant warnings as false positives [67, 141] might contribute to the low number of experienced false positives while using ESLint. More research should be invested in how developers perceive false positives and whether customizability is indeed a major factor in facilitating a good user experience with ASATs.

#### 7.2.3.6 Reasons for Not Using a Linter

Similarly to Johnson *et al.* [108], we have studied the challenges in using a linter by interviewing active users of such a tool. Furthermore, much like Christakis and Bird [74], we surveyed linter users and asked which challenges they face. In our survey, most participants had used a linter but we do not know how actively they used one or if they continued using a linter. As the main focus of our survey was to explore the experiences of linter users, we did not ask many questions to the few participants that had never used one. It would be informative to specifically target developers that do not use a linter or those that previously used a linter but have ceased using one. These developers might provide different insights into the challenging parts of using such a tool.

#### 7.2.3.7 Compare to Other ASATs

The main contribution of this thesis is that it increases our knowledge on why and how JavaScript developers use linters. These results should be further used to make a thorough comparison between how developers use these tools for JavaScript and how they use ASATs for other languages, *e.g.*, statically typed languages such as Java. Doing so would provide the research community with even more value to understand the general requirements for ASATs in the real world.

## 7.3 Final Conclusion

In this study we have applied three research methods where we first interviewed 15 experts on using ESLint, then analyzed over 9,500 ESLint configuration files from GitHub projects and finally surveyed more than 300 JavaScript developers about their experiences with using a linter, using the findings from the previous two analyses. The results from the three corresponding chapters complement each other, where the original results were further explained and verified in the last chapter.

With all three research methods we see that most developers use a linter to maintain code consistency. Nevertheless, both interview and survey participants claim that rules that relate to possible errors are far more important than those that relate to stylistic issues. Regarding how developers configure linters, they generally prefer to use a preset rather than to manually choose every rule that is used for a project. However, most developers additionally apply some configurations that are specific to a project or a team, especially rules that relate to stylistic issues. While it is important to be able to identify errors early in the development process by using a linter, even more developers feel the need to have consistent formatting where everyone taking part in a project uses quotes, semicolons and indentation in the same way. However, upholding these configurations can be a challenging task where it can be difficult to agree with teammates on which rules should be used for a project. Other aspects that have been reported in literature to be problematic with using linters and ASATs in general, namely the presence of false positives and the lack of dynamic analysis, do not appear to be perceived as challenges by most JavaScript developers.

The results of this study produced valuable implications for developers, tool makers and researchers, who can use the information to make better use of linters, to improve future versions of linters and to further research this important aspect of software development. The results and implications only directly apply for ESLint which is currently the most commonly used JavaScript linter. Nevertheless, a large part of the results can also be put into use for other linters to understand the general needs and motivation behind using a linter for JavaScript, and even for using a static analysis tool for other languages as well. Finally, as has been requested by many of the study's participants, we intend to summarize the findings and display them to the JavaScript community. With that we hope to receive feedback on our work along with providing the community with this valuable information and advice on using and developing linters for JavaScript.

# Bibliography

[1] AngularJS. https://angularjs.org. [Online; accessed 13-June-2017].

[2] Babel. https://babeljs.io. [Online; accessed 13-June-2017].

[3] babel-eslint via npm. https://npmjs.com/package/babel-eslint. [Online; accessed 2-June-2017].

[4] BigQuery. https://cloud.google.com/bigquery. [Online; accessed 29-May-2017].

[5] Checkstyle. http://checkstyle.sourceforge.net. [Online; accessed 11-July-2017].

[6] Closure Compiler. https://developers.google.com/closure/compiler. [Online; accessed 10-July-2017].

[7] Echo JS. http://echojs.com. [Online; accessed 14-June-2017].

[8] EditorConfig. http://editorconfig.org. [Online; accessed 12-July-2017].

[9] Electron. https://electron.atom.io. [Online; accessed 13-June-2017].

[10] Ember. https://emberjs.com. [Online; accessed 13-June-2017].

[11] ESLint. http://eslint.org. [Online; accessed 11-July-2017].

[12] eslint-config-airbnb-base via npm. https://npmjs.com/package/eslint-config-airbnb-base. [Online; accessed 29-May-2017].

[13] eslint-config-airbnb via npm. https://npmjs.com/package/eslint-config-airbnb. [Online; accessed 29-May-2017].

[14] eslint-config-ctrl via npm. https://npmjs.com/package/eslint-config-ctrl. [Online; accessed 17-June-2017].

[15] eslint-config-formidable via GitHub. `https://github.com/FormidableLabs/eslint-config-formidable`. [Online; accessed 17-June-2017].

[16] ESLint first major release (v1.0.0). `https://github.com/eslint/eslint/releases?after=v1.0.0`. [Online; accessed 23-May-2017].

[17] ESLint first release (v0.0.2). `https://github.com/eslint/eslint/releases?after=v0.1.0`. [Online; accessed 23-May-2017].

[18] eslint-plugin-flow via npm. `https://npmjs.com/package/eslint-plugin-flow`. [Online; accessed 5-June-2017].

[19] eslint-plugin-html via npm. `https://npmjs.com/package/eslint-plugin-html`. [Online; accessed 5-June-2017].

[20] eslint-plugin-import via npm. `https://npmjs.com/package/eslint-plugin-import`. [Online; accessed 29-May-2017].

[21] eslint-plugin-import via npm. `https://npmjs.com/package/eslint-plugin-import`. [Online; accessed 5-June-2017].

[22] eslint-plugin-jsx-a11y via npm. `https://npmjs.com/package/eslint-plugin-jsx-a11y`. [Online; accessed 29-May-2017].

[23] eslint-plugin-mocha via npm. `https://npmjs.com/package/eslint-plugin-mocha`. [Online; accessed 5-June-2017].

[24] eslint-plugin-react via npm. `https://npmjs.com/package/eslint-plugin-react`. [Online; accessed 29-May-2017].

[25] ESLint via GitHub. `https://github.com/eslint/eslint`. [Online; accessed 2-June-2017].

[26] ESLint via npm. `https://npmjs.com/package/eslint`. [Online; accessed 2-June-2017].

[27] Espree. `https://github.com/eslint/espree`. [Online; accessed 2-June-2017].

[28] Estraverse via GitHub. `https://github.com/estools/estraverse`. [Online; accessed 2-June-2017].

[29] FindBugs. `http://findbugs.sourceforge.net`. [Online; accessed 11-July-2017].

[30] Flow. `http://flowtype.org`. [Online; accessed 10-July-2017].

[31] GHTorrent. `http://ghtorrent.org`. [Online; accessed 29-May-2017].

[32] Google Hangouts. `https://hangouts.google.com`. [Online; accessed 23-June-2017].

[33] Impress.js-Tutorial via GitHub (example of a repository containing code for a tutorial). `https://github.com/cubewebsites/Impress.js-Tutorial`. [Online; accessed 23-May-2017].

[34] JavaScript Lint. `http://javascriptlint.com`. [Online; accessed 7-August-2017].

[35] jQuery. `https://jquery.com`. [Online; accessed 13-June-2017].

[36] JSCS. `http://jscs.info`. [Online; accessed 11-July-2017].

[37] JSHint. `http://jshint.com`. [Online; accessed 11-July-2017].

[38] JSLint. `http://jslint.com`. [Online; accessed 11-July-2017].

[39] JSLint on Google+. `https://plus.google.com/communities/104441363299760713736`. [Online; accessed 21-June-2017].

[40] JSLint via GitHub. `https://github.com/douglascrockford/JSLint`. [Online; accessed 21-June-2017].

[41] MQTT.js via GitHub (example of a repository containing four linters). `https://github.com/mqttjs/MQTT.js`. [Online; accessed 26-May-2017].

[42] MQTT.js via GitHub (example of a repository containing four linters, history showing where linters were removed). `https://github.com/mqttjs/MQTT.js/commit/a690c274e65bb708c2399d04af07b6f4618cc9be#diff-b9cfc7f2cdf78a7f4b91a753d10865a2`. [Online; accessed 26-May-2017].

[43] Node.js. `https://nodejs.org`. [Online; accessed 13-June-2017].

[44] npm. `https://npmjs.com`. [Online; accessed 13-June-2017].

[45] Octoverse 2016. `https://octoverse.github.com`. [Online; accessed 13-June-2017].

[46] PMD. `https://pmd.github.io`. [Online; accessed 11-July-2017].

[47] PMD Design, description of God Class metric. `http://pmd.sourceforge.net/pmd-5.0.1/rules/java/design.html`. [Online; accessed 24-June-2017].

[48] Prettier. `https://prettier.io`. [Online; accessed 12-July-2017].

[49] React. `https://facebook.github.io/react`. [Online; accessed 13-June-2017].

[50] React first release. `https://github.com/facebook/react/releases?after=v0.3.3`. [Online; accessed 5-June-2017].

[51] React Native. `https://facebook.github.io/react-native`. [Online; accessed 13-June-2017].

[52] SpiderMonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey. [Online; accessed 14-June-2017].

[53] Standard JS. https://standardjs.com. [Online; accessed 11-July-2017].

[54] stream-handbook via GitHub (example of a repository containing a programming guide). https://github.com/substack/stream-handbook. [Online; accessed 23-May-2017].

[55] SurveyGizmo. http://surveygizmo.com. [Online; accessed 20-June-2017].

[56] TraceMonkey. https://wiki.mozilla.org/JavaScript:TraceMonkey. [Online; accessed 13-June-2017].

[57] TSLint. https://palantir.github.io/tslint. [Online; accessed 12-July-2017].

[58] tslint-microsoft-contrib via GitHub. https://github.com/Microsoft/tslint-microsoft-contrib. [Online; accessed 17-June-2017].

[59] TypeScript. http://typescriptlang.org. [Online; accessed 10-July-2017].

[60] V8. https://developers.google.com/v8. [Online; accessed 13-June-2017].

[61] XO via GitHub. https://github.com/sindresorhus/xo. [Online; accessed 12-July-2017].

[62] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16(4):487–513, 2011.

[63] Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *ACM SIGPLAN Notices*, volume 49, pages 17–31. ACM, 2014.

[64] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE software*, 25(5), 2008.

[65] Nathaniel Ayewah and William Pugh. A report on a survey and study of static analysis users. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 1–5. ACM, 2008.

[66] Nathaniel Ayewah and William Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 241–252. ACM, 2010.

[67] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

[68] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 931–940. IEEE, 2013.

[69] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481. IEEE, 2016.

[70] Tim Berners-Lee, Robert Cailliau, Jean-François Groff, and Bernd Pollermann. World-wide web: The information universe. *Internet Research*, 20(4):461–471, 2010.

[71] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[72] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[73] Barry W Boehm. *Software engineering economics*, volume 197. Prentice-hall Englewood Cliffs (NJ), 1981.

[74] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343. ACM, 2016.

[75] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. Technical report, Wisconsin Univ-Madison Dept of Computer Sciences, 2006.

[76] Ravi Chugh, Jeffrey A Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. *ACM Sigplan Notices*, 44(6):50–62, 2009.

[77] Cesar Couto, Joao Eduardo Montandon, Christofer Silva, and Marco Tulio Valente. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Software Quality Journal*, 21(2):241–257, 2013.

[78] John W Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2013.

[79] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[80] Rafael Maiani de Mello, Pedro Corrêa Da Silva, and Guilherme Horta Travassos. Investigating probabilistic sampling approaches for large-scale surveys in software engineering. *Journal of Software Engineering Research and Development*, 3(1):8, 2015.

[81] David De Vaus. *Surveys in social research.* Routledge, 2013.

[82] ESLint. About. `http://eslint.org/docs/about`. [Online; accessed 23-May-2017].

[83] ESLint. Architecture. `http://eslint.org/docs/developer-guide/architecture`. [Online; accessed 2-June-2017].

[84] ESLint. Configuring ESLint. `http://eslint.org/docs/user-guide/configuring`. [Online; accessed 27-May-2017].

[85] ESLint. Rules. `http://eslint.org/docs/rules`. [Online; accessed 2-June-2017].

[86] Amin Milani Fard and Ali Mesbah. JSNOSE: Detecting JavaScript code smells. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 116–125. IEEE, 2013.

[87] Arlene Fink. *The survey handbook*, volume 1. Sage, 2003.

[88] David Flanagan. *JavaScript: the definitive guide.* O'Reilly Media, Inc., 2006.

[89] Zheng Gao, Christian Bird, and Earl T Barr. To type or not to type: quantifying detectable bugs in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering*, pages 758–769. IEEE Press, 2017.

[90] GitHub. About Stars. `https://help.github.com/articles/about-stars`. [Online; accessed 6-March-2017].

[91] GitHub. Language Trends on GitHub. `https://github.com/blog/2047-language-trends-on-github`. [Online; accessed 13-June-2017].

[92] GitHub. The most starred JavaScript projects. `https://github.com/search?l=JavaScript&q=stars%3A%3E1&type=Repositories&utf8=%E2%9C%93`. [Online; accessed 23-June-2017].

[93] Barney G Glaser and Judith Holton. Remodeling grounded theory. In *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*, volume 5, 2004.

[94] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. DLint: Dynamically checking bad coding practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 94–105. ACM, 2015.

[95] Danny Goodman. *JavaScript bible.* John Wiley & Sons, 2007.

[96] Georgios Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[97] Salvatore Guarnieri and Benjamin Livshits. GULFSTREAM: Staged Static Analysis for Streaming JavaScript Applications. *WebApps*, 10:6–6, 2010.

[98] Sarah Heckman and Laurie Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 41–50. ACM, 2008.

[99] Siw Elisabeth Hove and Bente Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *Software metrics, 2005. 11th ieee international symposium*, pages 10–pp. IEEE, 2005.

[100] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[101] Ecma international. ECMA-262, ECMAScript: A general purpose, cross-platform programming language. `https://ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf`, 1997.

[102] Ecma international. ECMA-262, ECMAScript Language Specification, 5th Edition. `http://ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262%205th%20edition%20December%202009.pdf`, 2009.

[103] Ecma international. ECMA-262, ECMAScript 2015 Language Specification, 6th Edition. `https://ecma-international.org/ecma-262/6.0/ECMA-262.pdf`, 2015.

[104] Ecma international. ECMA-262, ECMAScript 2065 Language Specification, 7th Edition. `https://ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, 2016.

[105] Ciera Jaspan, I Chen, Anoop Sharma, et al. Understanding the value of program analysis tools. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 963–970. ACM, 2007.

[106] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 59–69. ACM, 2011.

[107] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *SAS*, volume 9, pages 238–255. Springer, 2009.

[108] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[109] Stephen C Johnson. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977.

[110] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.

[111] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 121–132. ACM, 2014.

[112] Mark Kasunic. Designing an effective survey. Technical report, DTIC Document, 2005.

[113] Barbara A Kitchenham and Shari Lawrence Pfleeger. Principles of survey research: part 1: turning lemons into lemonade. *ACM SIGSOFT Software Engineering Notes*, 26(6):16–18, 2001.

[114] Barbara A Kitchenham and Shari Lawrence Pfleeger. Principles of survey research part 2: designing a survey. *ACM SIGSOFT Software Engineering Notes*, 27(1):18–20, 2002.

[115] Barbara A Kitchenham and Shari Lawrence Pfleeger. Principles of survey research: part 3: constructing a survey instrument. *ACM SIGSOFT Software Engineering Notes*, 27(2):20–24, 2002.

[116] Barbara A Kitchenham and Shari Lawrence Pfleeger. Principles of survey research part 4: questionnaire evaluation. *ACM SIGSOFT Software Engineering Notes*, 27(3):20–23, 2002.

[117] Barbara A Kitchenham and Shari Lawrence Pfleeger. Principles of survey research: part 5: populations and samples. *ACM SIGSOFT Software Engineering Notes*, 27(5):17–20, 2002.

[118] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, 28(8):721–734, 2002.

[119] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. Practically tunable static analysis framework for large-scale JavaScript applications (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 541–551. IEEE, 2015.

[120] Anton Kovalyov. Why I forked JSLint to JSHint. `https://web.archive.org/web/20130703100218/http://anton.kovalyov.net:80/2011/02/20/why-i-forked-jslint-to-jshint`, 2011. [Online; accessed 11-July-2017].

[121] Paul Krill. JavaScript creator ponders past, future. *InfoWorld. http://infoworld.com/article/08/06/23/eich-javascript-interview_1.html. Diakses pada*, 19, 2008.

[122] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[123] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, volume 10, 2012.

[124] Johan Linåker, Sardar Muhammad Sulaman, Rafael Maiani de Mello, and Martin Höst. Guidelines for conducting surveys in software engineering. 2015.

[125] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

[126] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 499–509. ACM, 2013.

[127] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359. IEEE, 2004.

[128] Tommi Mikkonen and Antero Taivalsaari. Using JavaScript as a real programming language. *Sun Microsystems, Inc.*, 2007.

[129] Frolin S Ocariza Jr, Karthik Pattabiraman, and Benjamin Zorn. JavaScript errors in the wild: An empirical study. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 100–109. IEEE, 2011.

[130] Stack Overflow. Developer Survey Results 2017. `https://insights.stackoverflow.com/survey/2017`. [Online; accessed 11-August-2017].

[131] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 161–170. IEEE, 2015.

[132] Joonyoung Park, Inho Lim, and Sukyoung Ryu. Battles with false positives in static analysis of JavaScript web applications in the wild. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 61–70. ACM, 2016.

[133] Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: Dynamic type inconsistency analysis for javascript. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 314–324. IEEE Press, 2015.

[134] Teade Punter, Marcus Ciolkowski, Bernd Freimut, and Isabel John. Conducting online surveys in software engineering. In *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*, pages 80–88. IEEE, 2003.

[135] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. *WebApps*, 10:3–3, 2010.

[136] Axel Rauschmayer. *Speaking JavaScript: An In-Depth Guide for Programmers*. O'Reilly Media, Inc., 2014.

[137] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 52–78. Springer, 2011.

[138] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *ACM Sigplan Notices*, volume 45, pages 1–12. ACM, 2010.

[139] Nick Rutar, Christian B Almazan, and Jeffrey S Foster. A comparison of bug finding tools for Java. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 245–256. IEEE, 2004.

[140] Amir Saboury, Pooya Musavi, Foutse Khomh, and Giulio Antoniol. An empirical study of code smells in JavaScript projects. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 294–305. IEEE, 2017.

[141] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 598–608. IEEE, 2015.

[142] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. In *ACM SIGPLAN Notices*, volume 48, pages 165–174. ACM, 2013.

[143] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, 25(4):557–572, 1999.

[144] Charles Severance. JavaScript: Designing a language in 10 days. *Computer*, 45(2):7–8, 2012.

[145] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. Improving developer participation rates in surveys. In *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*, pages 89–92. IEEE, 2013.

[146] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. *European Conference on Object-Oriented Programming (ECOOP)*, pages 435–458, 2012.

[147] Igor Steinmacher, Tayana Conte, Marco Aurélio Gerosa, and David Redmiles. Social barriers faced by newcomers placing their first contribution in open source software projects. In *Proceedings of the 18th ACM conference on Computer supported cooperative work & social computing*, pages 1379–1392. ACM, 2015.

[148] Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symposium on Programming (ESOP)*, volume 3444, pages 408–422. Springer, 2005.

[149] Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[150] Marco Torchiano, Daniel Méndez Fernández, Guilherme Horta Travassos, and Rafael Maiani de Mello. Lessons learnt in conducting survey research. In *Proceedings of the 5th International Workshop on Conducting Empirical Studies in Industry*, pages 33–39. IEEE Press, 2017.

[151] Kristín Fjóla Tómasdóttir. Codes derived from interview data. `https://doi.org/10.5281/zenodo.842207`. [Online; accessed 23-June-2017].

[152] Kristín Fjóla Tómasdóttir. Codes derived from survey data. `http://doi.org/10.5281/zenodo.824909`. [Online; accessed 23-June-2017].

[153] Kristín Fjóla Tómasdóttir. ESLint-config-analyzer. `https://github.com/kristinfjola/eslint-config-analyzer`. [Online; accessed 16-August-2017].

[154] Kristín Fjóla Tómasdóttir. JavaScript linter survey. `http://doi.org/10.5281/zenodo.814802`. [Online; accessed 20-June-2017].

[155] Kristín Fjóla Tómasdóttir. Linter interview visualisation example. `http://doi.org/10.5281/zenodo.817405`. [Online; accessed 23-June-2017].

[156] Kristín Fjóla Tómasdóttir. Post promoting survey on JavaScript user group (Iceland) on Facebook. `https://facebook.com/groups/nodejsis/permalink/1709121425832578`. [Online; accessed 14-June-2017].

[157] Kristín Fjóla Tómasdóttir. Post promoting survey on Reddit. `https://redd.it/6eumsp`. [Online; accessed 14-June-2017].

[158] Kristín Fjóla Tómasdóttir. Post promoting survey on Twitter. `https://twitter.com/kristinfjolato/status/871986432952479747`. [Online; accessed 14-June-2017].

[159] Kristín Fjóla Tómasdóttir, Maurício Aniche, and Arie van Deursen. Why and how JavaScript developers use linters. *To appear.* In *Proceedings of the 32st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2017.

[160] Paul Vorbach. npm-stat, download statistics for package react. `https://npm-stat.com/charts.html?package=react&from=2015-01-01&to=2017-05-31`. [Online; accessed 5-June-2017].

[161] Paul Vorbach. npm-stat, download statistics for packages eslint-config-airbnb, eslint-config-airbnb-base. `https://npm-stat.com/charts.html?package=eslint-config-airbnb&package=eslint-config-airbnb-base&from=2015-01-01&to=2017-05-31`. [Online; accessed 3-June-2017].

[162] Paul Vorbach. npm-stat, download statistics for packages eslint, jshint, jslint, jscs, standard. `https://npm-stat.com/charts.html?package=eslint&package=jshint&package=jslint&package=jscs&package=standard&from=2015-01-01&to=2017-05-31`. [Online; accessed 2-June-2017].

[163] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. Comparing bug finding tools with reviews and tests. *Lecture Notes in Computer Science*, 3502:40–55, 2005.

[164] Fadi Wedyan, Dalal Alrmuny, and James M Bieman. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 141–150. IEEE, 2009.

[165] James Q Wilson and George L Kelling. Broken windows. *Critical issues in policing: Contemporary readings*, pages 395–407, 1982.

[166] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[167] Nicholas C Zakas. *Maintainable JavaScript: Writing Readable Code*. O'Reilly Media, Inc., 2012.

[168] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 334–344. IEEE Press, 2017.

[169] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P Hudepohl, and Mladen A Vouk. On the value of static analysis for fault detection in software. *IEEE transactions on software engineering*, 32(4):240–253, 2006.

# Appendix A

# Interview Questions

The following is the list of questions that was asked in each interview.

## A.1 Participant Information

1. How many years/months of experience do you have as a professional developer?
2. How many years/months of experience do you have developing in JavaScript?
3. For how many years/months have you been a contributor to the X project?
4. What is your role in the X project? (e.g. *Founder, Lead Developer (core team), Maintainer/Developer, Tester, Documenter, Translator..)*

## A.2 Linter Usage

1. Why do you use a linter in your project?
2. How do you create your `.eslintrc` configuration file and maintain it? that is, how do you choose and prioritize the rules?
3. Given the rule categories from ESLint, which categories do you consider to be the most important and why?
4. Given the same categories, which categories do you consider to be the least important and why?
5. Which individual rules (within any category) do you consider to be the most important and why? (*e.g.*, top five rules)
6. Do you have any particular reasons for not choosing some of the rules for the configuration file in your project?
7. Do you use warnings and errors for different purposes?

8. Why are some files ignored in the `.eslintignore` file?
9. Are there any specific challenges about using a linter?
10. Do you experience false positives? if so, which?
11. With JavaScript being a dynamic language, do you feel that some features are missing in a static analysis tool such as ESLint?
12. If ESLint rules were to be prioritized in some manner, *e.g.*, to create a top 20 list of "must-have rules", which (if any) method(s) would you consider to be useful?
    a) Common configs from projects
    b) Developers' opinions of importance
    c) Most commonly eliminated errors
    d) The effects of the warnings/errors on change- and defect-proneness of files
13. Anything else about linters you would like to add?

# Appendix B

# Data Collection Query

## B.1 Query

Listing B.1 shows the SQL query that was used to retrieve the dataset for mining configuration files in Chapter 5. The query was executed on May 24th, 2017 using Google BigQuery, on the GHTorrent dataset for GitHub projects from April 1st, 2017. The actual query used to retrieve the data differs slightly since Google BigQuery does not allow to download result files that are above a certain limit in size. Therefore the dataset was queried in batches of 15,000 entries.

```sql
SELECT p.name, p.url, c.created_at, COUNT(w.user_id) AS stars, u.login
FROM [ghtorrent-bq:ght_2017_04_01.projects] AS p
  INNER JOIN (SELECT project_id, MAX(created_at) AS last_commit
    FROM [ghtorrent-bq:ght_2017_04_01.commits] GROUP BY project_id) AS x
    ON p.id = x.project_id
  INNER JOIN [ghtorrent-bq:ght_2017_04_01.commits] c
    ON x.project_id = c.project_id AND x.last_commit = c.created_at
  INNER JOIN [ghtorrent-bq:ght_2017_04_01.watchers] AS w
    ON p.id = w.repo_id
  INNER JOIN [ghtorrent-bq:ght_2017_04_01.users] AS u
    ON p.owner_id = u.id
WHERE p.language='JavaScript' AND p.deleted is false AND p.forked_from
    is null
GROUP BY u.login, p.name, p.url, c.created_at
HAVING stars >= 10
ORDER BY stars DESC
```

Listing B.1: SQL query for retrieving projects