Reasons and Drawbacks of using Trivial *npm* Packages: The Developers' Perspective

Rabe Abdalkareem
Data-driven Analysis of Software (DAS) Lab
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
rab_abdu@encs.concordia.ca

ABSTRACT

Code reuse is traditionally seen as good practice. Recent trends have pushed the idea of code reuse to an extreme, by using packages that implement simple and trivial tasks, which we call 'trivial packages'. A recent incident where a trivial package led to the breakdown of some of the most popular web applications such as Facebook and Netflix, put the spotlight on whether using trivial packages should be encouraged. Therefore, in this research, we mine more than 230,000 *npm* packages and 38,000 JavaScript projects in order to study the prevalence of trivial packages. We found that trivial packages are common, making up 16.8% of the studied *npm* packages. We performed a survey with 88 Node.js developers who use trivial packages to understand the reasons for and drawbacks of their use. We found that trivial packages are used because they are perceived to be well-implemented and tested pieces of code. However, developers are concerned about maintaining and the risks of breakages due to the extra dependencies trivial packages introduce.

CCS CONCEPTS

•**Software and its engineering → Software libraries and repositories;** *Software maintenance tools;*

KEYWORDS

JavaScript; Node.js; Code Reuse; Empirical Studies

ACM Reference format:
Rabe Abdalkareem
. 2017. Reasons and Drawbacks of using Trivial *npm* Packages:
The Developers' Perspective. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 3 pages.
DOI: 10.1145/3106237.3121278

1 INTRODUCTION

Code reuse has been widely accepted to be an essential approach to achieve high-quality software in a timely and cost-efficient manner [3, 15, 20]. Therefore, it is no surprise that emerging platforms such as Node.js encourage reuse and do everything possible to facilitate the sharing of code, often delivered as packages or modules that are available on package management platforms, such as the Node Package Manager (*npm*) [6, 21]. However, there are many cases where code reuse has had negative effects, causing increased maintenance costs and even legal action[2, 14, 18, 23]. For example, in a recent incident code reuse of a Node.js package called left-pad, which was used by Babel, caused interruptions to some of the biggest Internet sites, e.g., Facebook and Netflix. Many referred to the incident as the case that 'almost broke the Internet' [16, 24, 26].

Although the real reason behind the left-pad incident was about *npm* allowing authors to unpublish packages (which has been resolved now [22]), it raised awareness to the bigger issue of taking on dependencies for trivial tasks that can be easily implemented [12]. Since then, there have been many discussions about the use of trivial packages [13]. Loosely defined, *a trivial package is a package that a developer can easily code him/herself and hence, is not worth taking on an extra dependency for.* Many developers agreed that developers should implement such functions themselves rather than taking on dependencies for trivial tasks. Other work showed that *npm* packages tend to have a large number of dependencies [7, 8] and highlighted that developers need to take care when taking on extra dependencies since some dependencies can grow exponentially [4].

So, the question is "why do developers resort to using a package for trivial tasks, such as checking if a variable is an array?" At the same time, other questions regarding how prevalent trivial packages are and what the potential drawbacks of using these trivial packages remain unanswered. This research performs an empirical study involving more than 230,000 *npm* packages and 38,000 JavaScript projects to examine how prevalent trivial packages are in *npm* and how widely they are used in Node.js projects. Our empirical study is qualitative in nature and is based on survey results from 88 Node.js developers to better understand why developers resort to using trivial packages.

Our findings indicate that of the 231,092 *npm* packages in our dataset, 16.8% of them are trivial packages. Moreover, of the 38,807 Node.js projects on GitHub, 10.9% of them depend directly on one or more trivial packages. Our survey results showed that developers use trivial packages since they provide them with well-implemented/tested code and increase productivity. At the same time, the increase in dependency overhead and the risk of breakage of their projects are the two most cited drawbacks. In addition to this student research competition paper, we provide more details about our study of using trivial *npm* packages in [1].

2 METHODOLOGY

To understand the usage of trivial package in *npm*, we performed a quantitative analysis, and conduct a user survey, to gain a qualitative insight into the developers' perceptions about trivial packages.
Data Set: We obtained Node.js packages from *npm* platform and projects that use *npm* packages from GitHub. We mined the latest

ESEC/FSE'17, Paderborn, Germany
2017. 978-1-4503-5105-8/17/09…$15.00
DOI: 10.1145/3106237.3121278

version of all the Node.js packages from *npm* as of May 5, 2016. We downloaded the source code of 252,996 packages. We also mined all the Node.js projects on GitHub. We determine that a project uses *npm* packages by looking for the `packages.json` file, which specifies (among other) the *npm* package dependencies used by the project. To eliminate dummy projects that may exist in GitHub, we choose non-forked applications with more than 100 commits and more than 2 developers. At the end, we were left with 38,807 projects using *npm* packages.

Definition of Trivial Packages: To determine what constitutes a trivial package, we conducted a survey, where we ask JavaScript developers what considered to be a trivial package. We devised an online survey that presented the source code of 16 randomly selected Node.js packages that range in size between 4 - 250 JavaScript lines of code. Participants were mainly asked to 1) indicate if they thought the *npm* package was a trivial package or not and 2) specify what indicators they use to determine a trivial package. We provided the survey participants with a loose definition of what a trivial package is, i.e., a package that contains code that they can easily code themselves, and is not worth taking on an extra dependency for. Based on the survey responses, we find that 79% of the marked as trivial packages have less than or equal 35 lines of code and find that 84% of the votes marked packages that have a total complexity value of 10 or lower to be trivial packages. Hence, we define trivial packages as $\left\{X_{LOC} \leq 35 \cap X_{Complexity} \leq 10\right\}$, where X_{LOC} represents the JavaScript LOC and $X_{Complexity}$ represents the McCabe's cyclomatic complexity of package X.

User Survey: To better understand the developers' perceptions of using trivial packages (i.e. the reasons and drawbacks). We first applied the definition of trivial packages on all *npm* packages in our dataset, and then we identified developers who use trivial packages in JavaScript projects. We sent an online survey to 1,055 developers who use trivial packages. We asked participants about their software development experience; and two open-ended questions about the advantage and disadvantage of using trivial packages. We received 88 responses (8.3% response rate). We performed a qualitative analysis, where we manually examined the answers for the open-ended questions to identify the main reasons for and drawbacks of using trivial packages. The survey participants reported that 83 of them work in industry (68) or as independent developers (15). The remaining 5 work as casual developers. The majority (67) of the participants have more than 5 year of experience, 14 have between 3-5 years and 7 have 1-3 years of experience.

3 RESULTS

3.1 How Prevalent are Trivial Packages?

We examined prevalence from two aspects: the first aspect is from *npm*'s perspective, where we are interested in knowing how many of the packages on *npm* are trivial. The second aspect considers the use of trivial packages in JavaScript projects.

For each package (252,996 *npm* packages), we calculated the number of JavaScript code lines and removed packages that had zero LOC, which left us with a final number of 231,092 packages. Then, we applied the our definition of trivial packages on 231,092 packages and count the number of *npm* packages that satisfy our definition of trivial packages. Out of the total 231,092 *npm* packages we mind, 38,845 (16.8%) packages are trivial packages.

We also examined the number of projects on GitHub that use trivial packages. To do so, we examined the `package.json` file, which contains all the dependencies that a project installs from *npm*. For each JavaScript project in our dataset (38,807), we parsed the JavaScript code and use regular expressions to detect the required dependency statements, which indicates that the project actually uses the package in its code. Finally, we measured the number of packages that are trivial in the set of packages used by the project. We found that of the 30,807 projects in our dataset, 4,256 (10.9%) use at least one trivial package.

3.2 Reasons of Using Trivial Packages

We found five reasons for using trivial packages. However, due to space limitations, we only present the top three reasons.

R1. Well-implemented & tested (54.6%): The most cited reason for using trivial packages is that they provide well-implemented and tested code. For example, participants P68 state: *"Tests already written, a lot edge cases captured [...]"*.

R2. Increased productivity (47.7%): The second most cited reason is the improved productivity that using trivial packages enables. For example, participants P13 state: *"[...] and it does save time to not have to think about how best to implement even the simple things."*

R3. Well-maintained code (9.1%): A less common, but cited reason for using trivial packages is the fact that the maintenance of the code need not to be performed by the developers themselves. For example, participant P45 states: *" a highly used trivial package is probable to be well maintained."*.

3.3 Drawbacks of Using Trivial Packages

We identified seven drawback of using trivial packages. However, due to space limitations, we only present the top three drawbacks.

I1. Dependency overhead (55.7%): The most cited drawback of using trivial packages is the increased dependency overhead, e.g., keeping all dependencies up to date and dealing with complex dependency chains, that developers need to bear [6]. For example, P41 states: *"[...] people who don't actively manage their dependency versions could [be] exposed to serious problems [...]"*.

I2. Breakage of applications (18.2%) Developers concern about the potential breakage of their application due to a specific package or version becoming unavailable. For example, in the left-pad issue, the main reason for the breakage was the removal of left-pad, P4 states: *"Obviously the whole 'left-pad crash' exposed an issue."*.

I3. Decreased performance (15.9%) Developers mentioned that incurring the additional dependencies slowed down the build and increased application installation times. For example, P34 states *"[...], slow installs; can make project noisy and unintuitive by attempting to cobble together too many disparate pieces instead of more targeted code."*.

4 RELATED WORK

Studies of Code Reuse. Much of prior research on code reuse has highlighted its multiple benefits, which include improving quality, development speed, and reducing development and maintenance costs [3, 15, 19, 20, 25]. On the other hand, the practice of reusing source code has some challenging drawbacks including the effort and resource required to integrate reused code [10]. Furthermore, a bug in the reused component could propagate to the target system [11].

Studies of Other Ecosystems. Analyzing the characteristics of ecosystems in software engineering has been attracting more and more attention [4, 5, 7, 9, 17]. For example, Witter *et al.* [27] investigated the evolution of the *npm* ecosystem in an extensive study that covered the dependencies between *npm* packages, download metrics and the usage of *npm* packages in real applications. One of their findings is that *npm* packages and updates of these packages is steadily growing. Also, more than 80% of packages have at least one direct dependency. While our research corroborates some of these findings, the main goal is to empirically investigate the phenomenon of using trivial packages, in particular in Node.js projects.

5 CONCLUSION

The goal of this research is to examine the prevalence, reasons for and drawbacks of using trivial packages. Our findings indicate that trivial packages are widely used in Node.js projects. We also find that the majority of developers do not oppose the use of trivial packages and the main reasons developers use trivial packages is because they are considered to be well-implemented and tested. However, they do cite the fact that the additional dependencies' overhead as a drawback of using these trivial packages.

REFERENCES

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why Do Developers Use Trivial Packages? An Empirical Case Study on npm. In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'17)*. ACM.

[2] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. 2017. On Code Reuse from StackOverflow: An exploratory study on Android apps. *Information and Software Technology* 88 (2017), 148–158.

[3] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. 1996. How Reuse Influences Productivity in Object-oriented Systems. *Commun. ACM* 39, 10 (October 1996), 104–116.

[4] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. IEEE Computer Society, 280–289.

[5] Remco Bloemen, Chintan Amrit, Stefan Kuhlmann, and Gonzalo Ordóñez Matamoros. 2014. Gentoo Package Dependencies over Time. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*. ACM, 404–407.

[6] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, 109–120.

[7] Alexandre Decan, Tom Mens, and Maelick Claes. 2016. On the Topology of Package Dependency Networks: A Comparison of Three Programming Language Ecosystems. In *Proccedings of the 10th European Conference on Software Architecture Workshops (ECSAW '16)*. ACM, Article 21, 4 pages.

[8] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems. In *Proccedings of the 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER '17)*. IEEE.

[9] Alexandre Decan, Tom Mens, Philippe Grosjean, and others. 2016. When GitHub Meets CRAN: An Analysis of Inter-Repository Package Dependency Problems. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, Vol. 1. IEEE, 493–504.

[10] Roberto Di Cosmo, Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. 2011. Supporting software evolution in component-based FOSS systems. *Science of Computer Programming* 76, 12 (2011), 1144–1160.

[11] Mehdi Dogguy, Stephane Glondu, Sylvain Le Gall, and Stefano Zacchiroli. 2011. Enforcing Type-Safe Linking using Inter-Package Relationships. *Studia Informatica Universalis.* 9, 1 (2011), 129–157.

[12] David Haney. 2016. NPM & left-pad: Have We Forgotten How To Program? http://www.haneycodes.net/npm-left-pad-have-we-forgotten-how-to-program/. (March 2016). (accessed on 08/10/2016).

[13] Hemanth.HM. 2015. One-line node modules -Issue#10- sindresorhus/ama. https://github.com/sindresorhus/ama/issues/10. (2015). (accessed on 08/10/2016).

[14] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe. 2012. Where Does This Code Come from and Where Does It Go? - Integrated Code History Tracker for Open Source Systems -. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, 331–341.

[15] Wayne C. Lim. 1994. Effects of Reuse on Quality, Productivity, and Economics. *IEEE Software* 11, 5 (1994), 23–30.

[16] Fiona Macdonald. 2016. A programmer almost broke the Internet last week by deleting 11 lines of code. &+#http://www.sciencealert.com/how-a-programmer-almost-broke-the-internet-by-deleting-11-lines-of-code. (March 2016). (accessed on 08/24/2016).

[17] Konstantinos Manikas. 2016. Revisiting software ecosystems research: a longitudinal literature study. *Journal of Systems and Software* 117 (2016), 84–103.

[18] Stephen McCamant and Michael D. Ernst. 2003. Predicting Problems Caused by Component Upgrades. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '03)*. ACM, 287–296.

[19] Audris Mockus. 2007. Large-Scale Code Reuse in Open Source Software. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS '07)*. IEEE Computer Society, 7–.

[20] Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. 2004. An Empirical Study of Software Reuse vs. Defect-Density and Stability. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, 282–292.

[21] npm. 2016. What is npm? — Node Package Managment Documentation. https://docs.npmjs.com/getting-started/what-is-npm. (July 2016). (accessed on 08/14/2016).

[22] The npm Blog. 2016. The npm Blog changes to npm's unpublish policy. http://blog.npmjs.org/post/141905368000/changes-to--unpublish-policy. (March 2016). (accessed on 08/11/2016).

[23] Heikki Orsila, Jaco Geldenhuys, Anna Ruokonen, and Imed Hammouda. 2008. Update propagation practices in highly reusable open source components. In *Proceedings of the 4th IFIP WG 2.13 International Conference on Open Source Systems (OSS '08)*. 159–170.

[24] Brian Rinaldi, TJ VanToll, and Cody Lindley. 2016. Is left-pad Indicative of a Fragile JavaScript Ecosystem? http://developer.telerik.com/featured/left-pad-indicative-fragile-javascript-ecosystem/. (March 2016). (accessed on 08/24/2016).

[25] Manuel Sojer and Joachim Henkel. 2010. Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments. *Journal of the Association for Information Systems* 11, 12 (2010), 868–901.

[26] Chris Williams. 2016. How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript. http://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos. (March 2016). (accessed on 08/24/2016).

[27] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A Look at the Dynamics of the JavaScript Package Ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, 351–361.