A Data Exploration of the Package Ecosystem NPM

Ejnar Sörensen

Uppsala University, Uppsala, Sweden

15-01-2021

Abstract. JavaScript is today one of the most popular programming languages and its package ecosystem, NPM, has grown to become the largest and most relied upon of all package managers. In this essay an exploration of NPM package data is performed, and findings show that the ecosystem is still growing, but also that NPM packages in general are heavily reliant on a small amount of the most popular packages, many of which were released several years ago. Most packages existing on the platform today are not used and are rarely updated, which may be cause of some concern as they may be possible signs of stagnation and/or contain unfixed vulnerabilities.

Keywords: NPM, JavaScript, Packages.

1 Introduction

1.1 JavaScript and NPM

Over the years, JavaScript has gone from being the language associated with simple client side tricks of the 90's to being the full fledged application development language it is today. All the while remaining backwards compatible and introducing new features to the language. Along with the growth of JavaScript and more elaborate client web solutions, NPM (Node Package Manager) has grown to become the largest of all software registries, with many millions of downloads being made daily. (About Npm, n.d.)

Users interact with NPM using the NodeJS CLI. With this comes commands such as install, publish, audit and more; used to install, create new NPM modules and to assess their security. Packages on NPM are generally open for public use and are created by members of the community or companies wishing to further the open source community. Creating an NPM package is simple; the only criteria being that the package has a unique name, a package.json-file containing dependencies and other metadata and some code that can be resolved as a JavaScript module, thus invokable by through the require or import keywords in JavaScript.

Upon publish a package can be updated, have new versions of it published, downloaded as well as have new metadata added to it signaling its status to users, that it is for example deprecated, meaning it is to be removed in the future.

Using packages is a virtue for developers who save time during their projects by not having to write as much code themselves. There are however drawbacks to using third party packages. One of which stem from the extra size that a package brings with it, increasing the final size of the project that the eventual web page visitors need to download and thereby increasing load times. Secondly, and perhaps more problematic is how packages open up for vulnerabilities or hard to fix third party bugs, potentially causing dependent web applications to crash, fail to load or in some cases even allow third parties to access user data. The most recent example of which that was caused by NPM dependencies was the issue of Leftpadding, a minor package that was abruptly unpublished following a dispute between its creator and the NPM moderators. The results of Leftpadding becoming unobtainable for dependents caused thousands of other packages to crash. (Zapata, Kula, Chinthanet, et al., 2018)

1.2 Research question

Despite the fact that using NPM is commonplace amongst JavaScript developers, there is reason to believe that many do not know much of the ecosystem and that there might be a need for a higher awareness of the potential risks that relying on packages can entail. (Tal, L., 2020) To provide more general information and answers for this purpose I will explore some areas of interest in an EDA (Explorative Data Study) using data provided by Libaries.io and take a closer look at the problem area of vulnerabilities related to the rate of update amongst packages, rate of update being the most readily available topic that is relatable to seasonality, the original topic provided for this essay.

1.3 Method

Behrens (1997) describes the goals of an EDA as the finding of patterns to assist understanding and the building of a richer mental model of a subject. An effective approach in scenarios when little is known of the topic at hand that can result in rich descriptions of the data. This is a suitable method of exploration since not much general data is published and few have explored the Libraries.io datasets.

2 Preprocessing

Since this study is restricted to NPM, all other package managers' data was filtered out of the dataset before processing, leaving us with 1 277 220 packages, dating from the inception of NPM in 2008 up until the 13th of January 2020. The data that consist of a static csv-file was then stored in an online bucket on Google Cloud Platform and read into a python project on a virtual machine for processing. Because of the size of each row only the data of relevance for this study was kept to improve the speed at which the data could be read. During this processing some issues with this dataset were found that will be passed on to Libraries.io in hopes that it will be improved upon in the future, ranging from misalignment of columns to possible missing values without reason and descriptions that are difficult to make sense of or are too vague. The missing values of certain columns restricted the analysis to the data explored in the results.

3 Results

3.1 Size

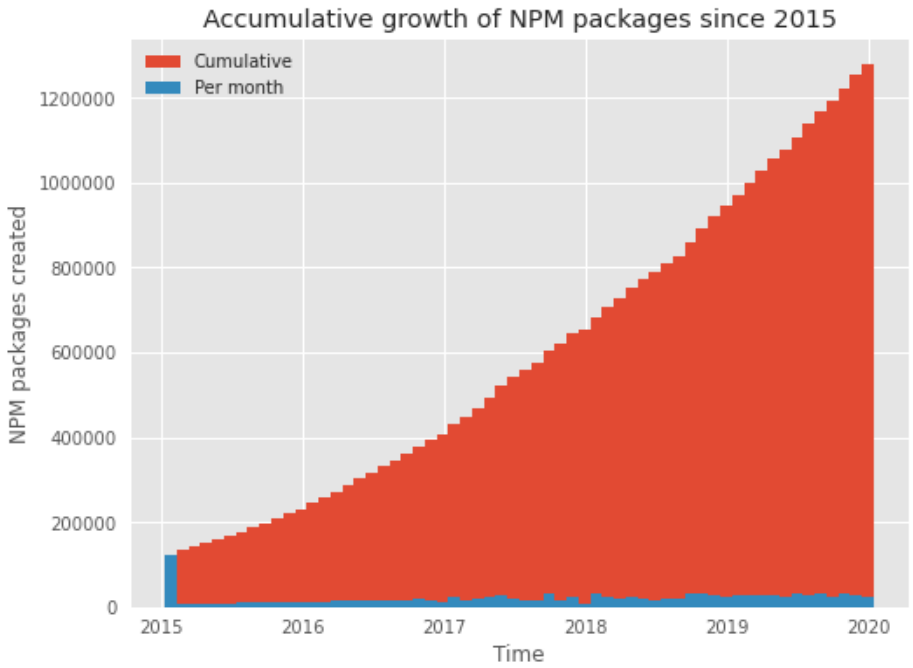

Fig 1. A bar chart showing the accumulative growth of NPM packages (red) and approximate per month releases (blue), from year 2015 to January 2020, based on the entire dataset provided by Libraries.io.

Looking at the release or creation date of all packages in the dataset we can determine that growth has been steadily increasing and that there are more than eight times as many packages in early 2020 than there were in 2015. Pre 2015 creation times are only available on projects with a repository on GitHub, for this reason packages posted pre 2015 are given a fixed creation date in the beginning of 2015.

3.2 Most popular projects

Taking a glance at the top 100 projects based on the number of other dependent NPM packages that each has, we can see that they generally consist of larger frameworks used for purposes of testing, bundling, correction of syntax and exporting newer JavaScript functionality to older browsers. Many of these are the types of packages that are added as development dependencies, meaning they do not need to be downloaded by clients when they are released on live websites.

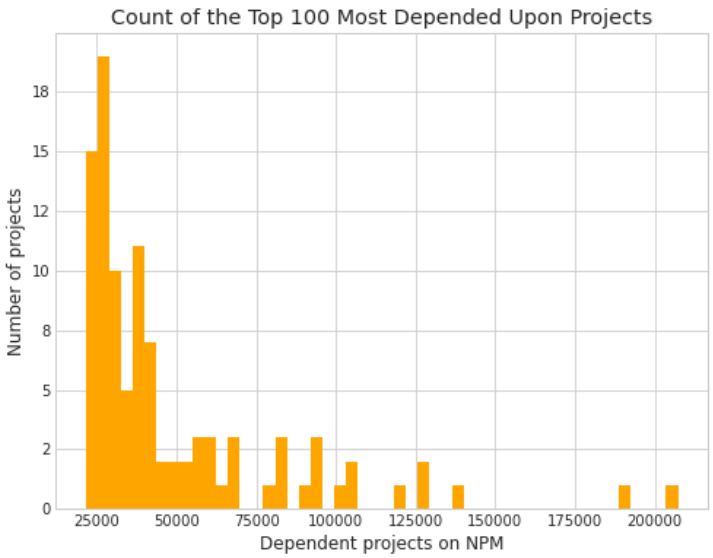

Fig 2. A bar chart showing the number of dependents that each of the top 100 popular projects has on NPM. Top referring to popularity in number of dependents.

The most depended upon package is Mocha, a test framework released in 2011. With 207k dependent NPM projects, Mocha is depended upon by approximately 16% of all NPM packages. After Mocha, Eslint, TypeScript and Webpack the number of dependents trickle down to 25-50k for a majority of the top one hundred. Most packages have fewer than this, however, with the median being 0 dependents and the average being 2,4 dependents when looking at all 1,2 million projects in the set. Comparing these two metrics we can tell that the dataset is top heavy in this regard, with most of the dependents coming from the top packages, and that the ecosystem consists in majority of projects that have 0 dependents.

JavaScript developers prefer smaller sized micro-packages compared to Python developers, who seemingly more often depend on larger tools for their packages. On JavaScript the average depth of a dependency tree is 4,39 packages deep, while on PyPI it is 1,71. This means that from the root package, the average dependency tree size is 86,55 nodes in a JavaScript project versus 7,33 on Python projcets. (Tal, L., 2020)

Fig 3. A bar chart showing the release year of the top 100 projects on NPM. Top referring to popularity in number of dependents.

Looking at the year of first release amongst the top 100 projects we can determine that none of the most depended upon packages were released in either 2018 or 2019. A reason for this could be that massing up a dependency base takes a long time, but it also shows that most of the very popular packages were first released over half a decade ago. Most of the top 100 were first released in 2014, with 2013 and 2015 being next in line.

3.3 Rate of update and activeness

With a larger portion of the set we can further compare the packages temporally and analyze their activity. By subtracting the date at which a package was first published on NPM with the date it was last updated according to the dataset, we get the time that has passed between publish and last update. Last update in this case referring to the date that the repository was last pushed to, a metric only available for GitHub repositories, meaning that n=775 789 packages for Fig 4. This interval of time can be understood as how long a package has been active for and give us a picture of how active packages from previous years are.

When this data was generated some discrepancies were found as some rows had negative values (n=3340), meaning their last recorded pushed to date was before their recorded publish date, as per the host. A likely reason for this is that the repository finished development before being published on NPM by its creator and has not been updated since. For readability of the graph these negative values are not displayed, though there are risks of inaccuracy in comparing these values when the true creation date is unavailable. Comparing the results with findings others have made show a similar trend.

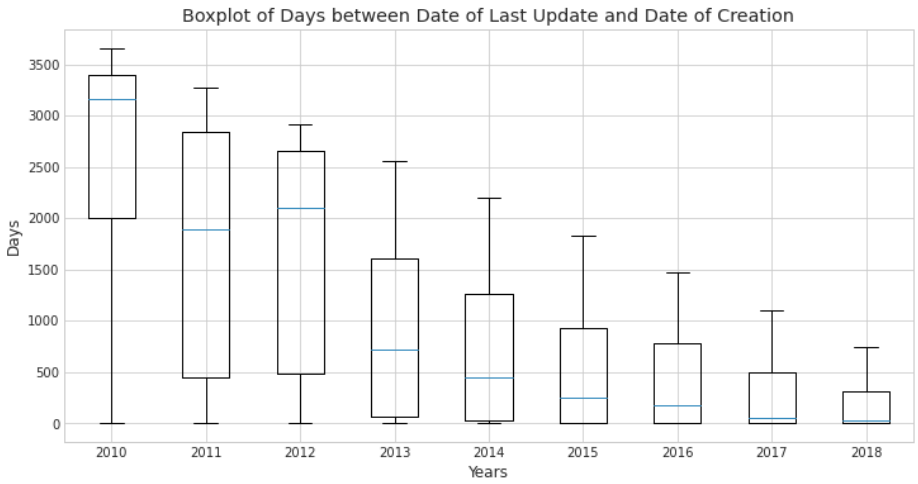

Fig 4. A boxplot showing the average time measured in days between creation or publish on NPM and its last recorded update. Contains only packages with a repository on GitHub.

Fig 4 indicates that amongst NPM packages with repositories on GitHub, the rate of activeness varies from year to year. The general trend that older packages have been updated for a longer period of time than newer ones is to be expected since they have had more time to become updated, but it appears that many packages from 2013 and on more rarely see updates. The average package released in 2014 has not been changed since approximately early 2016. This indicates that while there are many packages on NPM, most have not been active for a number of years.

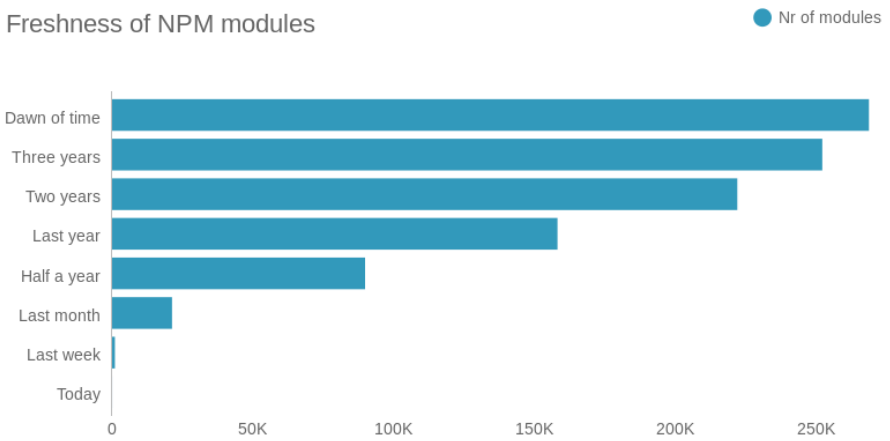

Fig 5. A bar chart by Düüna, K. (2018) showing the ‘Freshness of NPM modules’ in average time since a NPM package last update.

Looking at Fig 5, which compares the timestamp of last update date and a more recent date, made by Düüna, K. (2018), we see a similar trend as with Fig 4. The dataset used by Düüna was obtained by manually caching projects rather than downloading them from Libraries.io.

3.4 Package status

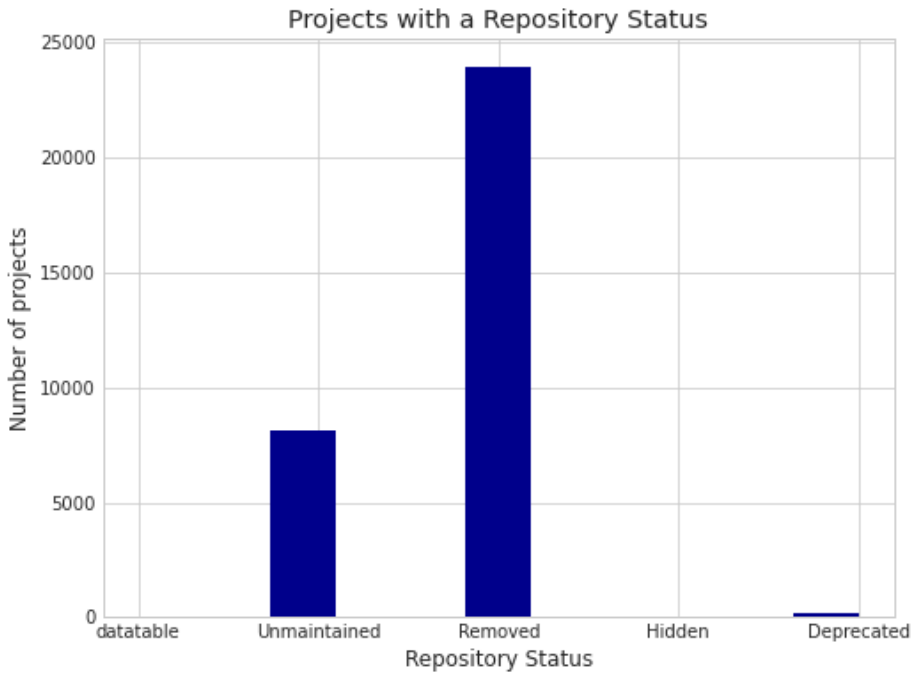

Fig 6. A bar chart displaying the repository statuses of NPM projects, with number of projects for each status present in the data.

Finally, coming back to packages marked with a status, we can see that the ones available on NPM are removed, deprecated or unmaintained. We observe 32279 (2,6%) of such packages out of the 1,2 million. Since this dataset covers only the latest version of a package, there are likely deprecated versions of a project that are not present on Fig 6. As most packages are not marked as removed, they are still accessible on NPM. Almost no final version of any project has been marked as deprecated, this may be more explainable however, since a deprecated package is meant to eventually be removed. (npm Docs) The unmaintained packages likely indicate that no further work will be made on them, a use on your own risk kind of caution. NPM provide a lengthy documentation for how their ecosystem is to be maintained, but offer no descriptions on the ‘unmaintained’ status keyword.

4 Discussion

In summary, some notable metrics amongst the results was that the number of packages on NPM is steadily increasing as of year 2020, more than ten years after its inception. Also, the use of these packages, counted in number of dependents, is not distributed equally across the set, but instead heavily skewed towards the top. Packages in general were determined to have few dependents and seldom see updates.

When compared with another popular package manager it was found from an external comparison that JavaScript developers have a wider and deeper tree of dependencies, where even though the average dependency count was 4,39 packages, the final count amounted to 86,55, taking every chain-depended upon package into account. (Tal, L., 2020) This type of dependency chain proved dangerous in the case of Leftpadding, where a simple package that helped align elements on a webpage brought a large portion of the web to its knees when it was unpublished. (Williams, C., 2020) Large NPM packages depended upon Leftpadding, which then propagated amongst themselves and on to smaller packages so that many in the end suffered the consequences and stopped working. When this amount of packages are reliant on the same packages the risk is increased, particularly when the large packages themselves are also reliant on smaller more volatile packages such as Leftpadding. It seems this type of incident is difficult to avoid in an ecosystem such as NPM because of how closely knit it is, and perhaps it should be up to the larger packages to take responsibility in choosing dependencies with more care if similar events are to be avoided in the future.

To better estimate what dangers there are in depending on third party packages, Decan, Mens and Constantinou (2018) in their study, found that many of the vulnerabilities found on the platform are in old packages. While most problems are found and fixed before or within six months of the release of a package, the remainder was found to rarely be taken care of and would be left in their vulnerable state. At the same time early results from a different study suggests that up to 73.3% of the clients in their test that depend on a vulnerable dependency were in fact safe from its threat since they were not using the affected function. (Zapata, Kula, Chinthanet et. al., 2018) As such, understanding the danger is difficult; although most vulnerabilities could be found in older packages it was not determinable that they would cause any harm.

Trying to put the problems into comparison with the benefits of NPM is not an easy task either, seeing as there, with a few major exceptions, have not been that many issues. Decan et. al (2018) also found that the epidemic following large scale vulnerabilities such as heartbleed, the infamous OpenSSL issue, and Leftpadding has caused a response from the industry where more automated security tools are being used to scan packages and developers running tools that add badges to their GitHub repositories, showing that they keep up to date with their dependencies. The effects of which have not been clear yet.

In this study it was found that many NPM packages are left without updates for long periods of time. One possible explanation for this could be that there is little incentive in continuing development on a project that sees little to no use. Most packages were found to be without dependents, and as such their creators may have left them as they

were when no one used them. This opens the question of findability of NPM packages, how users go about finding new packages, and if a reason that the popularity is skewed is because the old, already established packages are the only ones to be found. The top one hundred packages showed that the popular ones were all relatively old, and that there were no competitors from the last few years. Whether this is because of poor findability is something that may be of interest in future studies, even more so if data from coming years show the same trend. JavaScript is deemed growing and innovative, but if the package ecosystem stagnates it may affect the community.

Lastly, knowing that older packages without updates are more likely to involve risks, and that most packages are not updated, a discussion should perhaps be had about the status 'unmaintained' that approximately eight thousand packages had, and the 'deprecated' status that only a few hundred was seen to use in their final version of their projects. NPM themselves mention deprecation as a natural first step in the process of a package's removal. It is to be marked as deprecated to provide backwards compatibility so that dependents are given time to adjust and find a replacement, while being aware that the package will eventually become unavailable. (npm Docs.) Common reasons for deprecation can be age or that the feature becomes incorporated as a natural part of the language or that better packages with the same purpose replace it (Bates, C., 2019) The deprecated status may then be well reserved for projects to be removed, but the 'unmaintained' status should perhaps see more use. Generally, the term unmaintained would signify that a project is no longer being worked upon and that no further updates should be expected, but seeing as there is no clear definition, nor in fact any mention, of this status in the NPM documentation, could explain its unuse and raises the question if it is perhaps an error in the dataset by Libraries.io. Little is known of how aware developers are of handling packages on NPM but adopting a status keyword such as 'unmaintained' could prove helpful to users, letting them know that although nothing is necessarily wrong, the package is not actively being developed.

5 Reproducibility

For the sake of reproducibility the source code for the analysis made in this essay is publicly available on GitHub, with instructions of use and accessing of data.
https://github.com/enars/Data-exploration-of-NPM

The data used is provided by Libraries.io under the Creative Commons Attribution-ShareAlike 4.0 International License and can be downloaded and used for the appropriate purposes from https://libraries.io/data.

The file used is called **libraries-1.6.0-2020-01-12.tar.gz** was last accessed on 2021-01-15 by the author.

References

1. Zapata, R. E., Kula, R. G., Chinthanet, B., Ishio, T., Matsumoto, K., & Ihara, A. (2018, September). Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm JavaScript packages. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 559-563). IEEE.

2. Behrens, J. T. (1997). Principles and procedures of exploratory data analysis. *Psychological Methods*, *2*(2), 131.

3. Bates, C. (2019, July 28). What is node package deprecation? - Clyde Bates. Medium. https://medium.com/@a.bates1993/what-is-node-package-deprecation-82e42d40f46d

4. Düüna, K. (2018, May 18). What I learned from analysing 1.65M versions of Node.js modules in NPM. Medium. https://blog.nodeswat.com/what-i-learned-from-analysing-1-65m-versions-of-node-js-modules-in-npm-a0299a614318

5. Tal, L. (2020, July 7). How much do we really know about how packages behave on the npm registry? Snyk. https://snyk.io/blog/how-much-do-we-really-know-about-how-packages-behave-on-the-npm-registry/

6. Williams, C. (2020, June 28). How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript. The Register. https://www.theregister.com/2016/03/23/npm_left_pad_chaos/

7. npm Docs. (n.d.). Npmjs. Retrieved January 14, 2021, from https://docs.npmjs.com/about-npm/

8. Decan, A., Mens, T., & Constantinou, E. (2018, May). On the im-pact of security vulnerabilities in the npm package dependency network. In Proceedings of the 15th International Conference on Mining Software Repositories (pp. 181-191).