Effect of Project Dependencies on the Health of the Software Project

G.V.S Bhaskar
MS in Computer Science, NAU
vg588@nau.edu

Anudeep Uppu
MS in Computer Science, NAU
au282@nau.edu

Keerthana Vijaykumar
MS in Computer Science, NAU
kv582@nau.edu

Sahana Vallu
MS in Computer Science, NAU
sv799@nau.edu

Mohanthi Sandeep
MS in Computer Science, NAU
ma3924@nau.edu

Abstract

Software developers rely on an increasingly high number of packages to build their projects and this dependency on packages could lead to major issues if the packages are deprecated which in turn affects the health of the project. Hence, maintenance of these packages with the latest versions are crucial. Majority of the package developers offer package compatibility with one or two previous versions, and many new programs which are written for the current/latest version might not run on older versions. In this paper, we will deduce the health of the project by perceiving the deprecated dependencies. We used Git-hub Search GUI (GH Search) to retrieve repositories. For each repository, the contents of package-lock.json files were fetched from git-hub, and these packages further fetch dependency details like the deprecated packages and the latest version of the packages from the npm registry. From the obtained results, a comparison of the versions led to the following details – the upgradation count, the outdated count, the total dependency count and up-to-date version count. These results helped us answer our research questions. Our findings indicate that usage of deprecated packages can lead major issues faced by developers in terms of security issues and maintenance issues. It is explicit that there is a dependency of repositories on deprecated packages which affects the health of the project. In order to overcome these issues, frequent updating to newer version of package is mandatory.

CCS Concepts: • **Software and its engineering** → *Software creation and management*; **Maintaining software**.

Keywords: Software Management and Maintenance, Post-Development Issues, Dependency Management, Node Package Manager(NPM), JavaScript Projects.

ACM Reference Format:
G.V.S Bhaskar, Anudeep Uppu, Keerthana Vijaykumar, Sahana Vallu, and Mohanthi Sandeep. 2022. Effect of Project Dependencies on the Health of the Software Project. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The software development paradigm has led to a shift where code reuse has opened new opportunities for developers to build software, delivered in the form of reusable packages or modules that are available on the package management platforms such as node package manager (NPM), PIP, NuGet package manager, etc. Developers these days to a large extent rely on an increasingly high number of packages to build their projects, reusing code increases productivity, reduces the time-to-market, and improves software quality [8]. However, this model of development creates a huge dependency on these packages. At times, these packages could be deprecated, and managing these deprecated packages leads to serious issues which need to be addressed.

1.1 Problem

When developers or organizations encounter difficulties with deprecated packages present in the project or if the information regarding these versions was not informed to the developers/managers, this could lead to the issue of malfunctioning software in the immediate future.

1.2 Evidence

An incident of evidence is the eviction of the "left-pad" package that led to extensive damage among giant internet sites. This was relied upon by organizations such as Facebook, Netflix, and Airbnb, the problem was promptly fixed [1]. Another incident in NPM (Node Package Manager) is the release of a backward incompatible minor version 1.7.0 of the package "underscore" that caused many complaints among dependent packages about underscore not respecting Semantic Versioning (SemVer) [9].

An effective solution to manage dependencies is proposed as semantic versioning. This allows managers to receive upgrades and minor fixes of dependencies. The problematic issue is simply allowing dependencies to update automatically without notice. Although semantic versioning has been referenced in literature to overcome the difficulty in upgrading dependencies, these are usually referred to as other content' to explore issues like security vulnerabilities. However, previous research has shown that developers do not always conform to Semantic versioning. This created major problems due to outdated dependencies and breaking changes. Providing a solution for the existing problem can help the developers who use dependencies that could be deprecated. Acknowledging them with the latest versions or upgrades may result in better maintenance of the programs and systems.

1.3 Research Questions

RQ1: What is the health of project dependency involved in the project?

Firstly, the health of the project depends upon the projects involving deprecated/outdated packages. The project's dependency on this kind of package is inversely proportional to its health report. The insights gained concerning the outdated packages or the incompatible packages in this process of the study helped to identify the project's state of health.

RQ 2: What are the compatibility issues involved in the upgradation of project dependencies in the future?

When dependencies are updated to their latest version, developers or organizations might have problems if they are not notified about updated or outdated versions which might lead to software malfunctioning which poses severe problems in the future or right away. We should constantly monitor for changes to any kind of dependencies we utilize frequently, or else the product might not function appropriately. The maintenance of the product becomes critical if the dependencies are not up to date which in turn consumes the developer's time in identifying/resolving issues.

1.4 Purpose Statement

The main purpose of this analysis is to mitigate the problem faced by developers/naive programmers to resolve outdated/deprecated package dependency issues or incompatibility of the latest dependency version with the existing dependencies. It is important to identify the dependency because the majority of the projects depend upon already existing packages and if the packages get deprecated without prior information this could create a lot of dependency issues which could affect the project. In these kinds of situations, we need to completely change the packages which the project is depending on and might need to install new packages and the functionality of the packages might also change which might not satisfy the project requirements. By following this approach, we can achieve the purpose of providing the information to the end users which solves the issues related to deprecated package dependency. Henceforth, identifying the package dependency is crucial and needs to be taken care of.

1.5 Objective

The objective/goal of this study is to identify the impact of these deprecated packages and their internal packages on a project as well as compatibility issues occurring due to package upgrades. As mentioned above, programmers will not be intimated regarding the deprecated packages present in their project which could break the project. In this case, the code has to be completely modified or needs the inclusion of other/similar functionality. When the analysis is done on these packages it helps us study the health of the project and gain better insights regarding the current state of the project dependencies.

1.6 Scope

We would be restricting our scope to JavaScript/Typescript projects for the package dependency analysis and this study would be resolving up to 3-4 levels of package or internal package dependencies to identify the deprecated and latest version of project dependency.

Observing the evolution of the dependency issues over time, we conclude that these issues are being addressed, but new issues tend to occur more frequently than regular issues (issues that are being fixed). This led to an inclination in the problem being faced. The agenda of this paper is to develop a repository where developers or naive programmers can clone it to their local system to analyze their project dependency on deprecated or outdated packages.

2 Research Methods

2.1 Data Collection

Our preliminary data collection process began with a considerable number of repositories being gathered from Git-hub API. We could retrieve around 100 results per page as Git-hub API has restricted a maximum of 100 results per page. So, we were fetching 10 pages (equal to 1000 records). Unfortunately, we were unable to pull in all the 1000 records from the Git-hub API and only 926 records were fetched.

To avoid this, we used Git-hub Search GUI (GH Search), a tool that uses a server filtering technique that provides the exact number of repositories we request and makes querying much easier. In the GH Search tool, we applied a number of filters to create our required dataset. We chose the language as JavaScript since it is a popular language on git-hub and a date-based filter for repositories that were created between 2010 – to date. We further refined our search, by taking into consideration the number of stars as '3366' [10] and the license as 'MIT' by keeping in mind popularity, engagement, and recency as we are looking for popular repositories which resulted in 1130 repositories. As every JavaScript project consists of package.json and package-lock.json files, we chose to study projects containing these two files and found 1043 repositories related to them.

During our analysis, we observed that the packages present in the package.json file save the versions with signs (^,*, ~, <= , >=, <, >) showing that packages with these signs can support any higher version hence we narrowed down our search to package-lock.json files as they contain the exact snapshot versions. On the whole, we were left with 377 repositories containing the package-lock.json file.

For each repository, the contents of package-lock.json files were retrieved from git-hub, and these packages further fetch dependency details like the deprecated packages and the latest version of the packages from the NPM registry. Based on these results, for each of the packages obtained, we fetched the dependency information such as the package name and the corresponding current version and stored these results.

From the obtained results, a comparison of the versions led to the following details – **the upgradation count, the outdated count, the total dependency count, and the up-to-date version count**. These results helped us answer our research questions and analyze the health of project dependency.

2.2 Data Analysis

Our initial analysis started with calculating the mean values, the total filtered counts were 330 and the calculated mean of the 'outdated count' was 11.62, and the mean of the 'upgradation count' resulted in 350.31. We also sorted the data received based on star count and watchers count where if two or more repositories have the same star count, we sort them based on the watchers count. We observed a correlation between the upgradation count and total dependency count and a correlation between outdated packages and total dependency count. The relationship seems to be directly proportional, with the increase in the total number of dependencies the other two entities will also increase.

We further analyzed 111 packages on the whole and manually compared them to check whether these packages had any deprecated versions listed in the NPM registry. While comparing, we observed that majority of the repositories had the latest versions of the package. Among all these 111 packages we found out that only 2 packages had deprecated versions which the repositories were majorly dependent on.

Firstly, there was a package named 'fsevents' with the current version of 2.3.2. In total, 249 repositories were depending on this package. There were 51.4% of repositories which were depending on the deprecated version of this package. Secondly, there was a package named 'mkdirp' with the current version of 1.0.4. In total, 255 repositories were depending on this package and there were 45.34% of repositories which were depending on the deprecated version of this package.

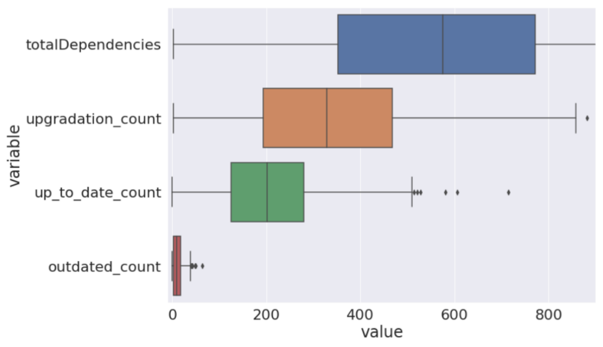

Figure 1

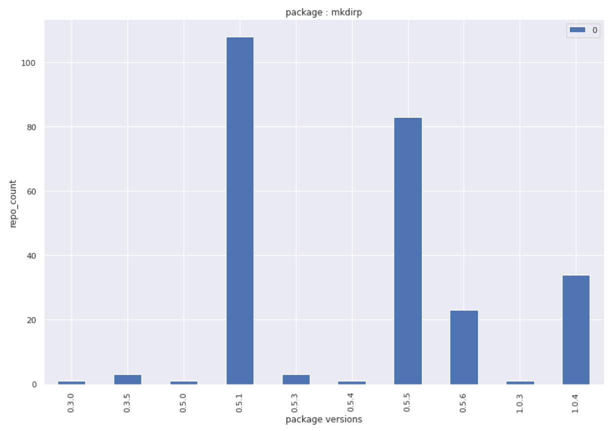

Figure 2

3 Results

From our analysis we found out that all the repositories are using the dependencies/packages in the range between 300 and 700. From Figure 1, we have observed that the average of the upgradation count is more when compared to up-to-date count. This means majority of the projects don't have the latest version of the packages which could lead to security/stability issues and hence it is recommended that packages should be updated frequently. The graph also shows that the outdated count is negligible which means that the projects depending on deprecated packages are less.

RQ1: How does deprecated packages affect the health of project dependencies?

The histogram depicts multiple versions for package 'mkdirp'. The latest version of the package is 1.0.4, we found out that 108 repositories are depending on the version 0.5.1 which is deprecated in NPM registry. This clearly describes that majority of the repositories are depending on the packages which are deprecated. Hence, updating packages up to date is considered as a crucial task.

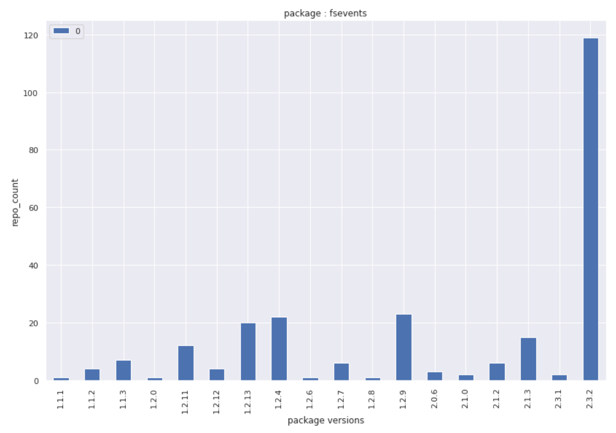

Figure 3

RQ2: How does the project health get affected if the dependencies are not upgraded to their latest versions? The histogram related to package 'fsevents' depicts that majority of the repositories have latest version of the package which is 2.3.2. However, the remaining 51.4% repositories are not using the latest version which might be missing the important feature i.e, Native access to MacOS FSEvents in Node.js - fast and lightweight alternative to kqueue. These repositories might have a severe effect on their project in a long run as native access to popular operating systems like Windows, Linux, Mac is much more needed. These repositories are expected to install the latest version of the package available to experience the latest functionality.

4 Related Work

There are a couple of works that are closely related to our research work since package dependencies and their related issues are quite widespread these days due to the high usage of packages in the software development process.

Managing package dependencies and their associated internal dependency is essential in the software ecosystem. In the study of C. Artho et al. stated that 80% of conflict defects were due to resource access, data, and the uncommon combination of packages[3]. These conflicts can be reduced by making the best testing combinations of packages that may conflict and checking the meta-data of the packages.

These package dependency issues may arise due to improper dependency upgrades. The results of the J. Cox et al. stated that systems using outdated dependencies are four times more likely to have security vulnerabilities than systems using up-to-date dependencies[5]. This has been explained using the concept of dependency freshness.

Another reason might be the backward incompatible updates leading to various issues. As discussed in the paper[6], these incompatibility issues can be resolved by using Semantic Versioning or semVer i.e, introducing a set of simple rules that suggest how to assign version numbers to inform developers about potentially breaking changes. But this version numbering has not shown a greater impact in resolving the dependency management issues. The paper[7] states that 1/3rd of the modules using advisory dependencies resolve to the vulnerable version and this is due to a lack of awareness or discussion about vulnerable/deleted dependencies.

Results presented in the paper[2] indicate that trivial packages are commonly and widely used in JavaScript and Python applications. In the dataset used for the study, 18.4% of the NPM and 2.9% of the PyPI trivial packages have more than 20 dependencies. In this paper, we have narrowed down our scope to JavaScript dependencies where the package dependency issues are more prominent.

The NPM dependency smells have a major impact on the docker images. A study that analyzed the 961 images from three official repositories using NODE.JS and 1099 security reports of packages stated that outdated NPM package dependencies introduced a higher risk of security vulnerabilities and suggested that docker maintainers should keep eir installed JavaScript package up-to-date[11].

The "left-pad incident"(worst dependency smell) stated in the paper[4], in which a package with 11-lines of code was removed from NPM, caused a significant downtime on major websites such as Facebook, Instagram, and LinkedIn. A similar study was performed on a dataset of 1146 active JavaScript projects in order to identify the dependency smells. Their findings revealed that these smells are prevalent and 80% of the projects are infected with two or more distinct smells[8]. This paper introduced a tool called Dependency Sniffer that analyzes a large number of projects and identifies the potential dependency smells. Along similar lines, this paper also focuses on identifying the outdated packages that need upgradation for the given JavaScript projects.

5 Threats to validity

Initially we tried to fetch the repositories from GitHub. Our requirement was to fetch 1000 repositories but unfortunately, we could get only 926 repositories. We then shifted to GH Search for fetching the repositories according to our requirement. Secondly the correlation of the results didn't match to our expectations. Hence, we moved to Qualitative data analysis instead of statistical analysis. We manually checked 111 packages in the npm registry and checked whether they had any deprecated versions and later we compared the project dependency on deprecated packages.

6 Implications

Our results imply that project dependency on deprecated packages could lead to serious security and compatibility issues. It is always suggested to update the packages regularly on time and experience the latest functionality.

7 Conclusion

Our objective was to identify the project dependency on deprecated packages and how these packages could effect the projects and could create serious impacts. We performed a study on 377 repositories to check for package deprecation dependency and also we manually checked for 111 packages to identify its deprecated versions. According to our results we observed that there is a severe dependency of projects on deprecated packages and not updating it frequently could lead to security and stability issues and the end user could also miss the latest functionality.

8 Future Work

We will be creating an GitHub extension which will analyze the dependency issues in a given JavaScript repository and shares the information of repository health to the project managers, team developers and other team members. Also, we would like to extend our scope to other popular programming languages and analyze the dependency smells.

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. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 385–395. https://doi.org/10.1145/3106237.3106267

[2] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. 2020. On the impact of using trivial packages: an empirical case study on npm and PyPI. *Empirical Software Engineering* 25 (3 2020), 1168–1204. Issue 2. https://doi.org/10.1007/s10664-019-09792-9

[3] C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, and S. Zacchiroli. 2012. Why do software packages conflict? *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, 141–150. https://doi.org/10.1109/MSR.2012.6224274

[4] Filipe Roseiro Cogo. 2020. https://dl.acm.org/doi/abs/10.5555/AAI28387722. https://dl.acm.org/doi/abs/10.5555/AAI28387722

[5] Joel Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. 2015. Measuring Dependency Freshness in Software Systems. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 109–118. https://doi.org/10.1109/ICSE.2015.140

[6] Alexandre Decan and Tom Mens. 2021. What Do Package Dependencies Tell Us About Semantic Versioning? *IEEE Transactions on Software Engineering* 47 (6 2021), 1226–1240. Issue 6. https://doi.org/10.1109/TSE.2019.2918315

[7] Joseph Hejderup. 2015. In Dependencies We Trust: How vulnerable are dependencies in software modules?

[8] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. 2021. Dependency Smells in JavaScript Projects. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3106247

[9] Lucas Mancini. 2018. A simple strategy to manage your JavaScript project's dependencies. https://medium.com/coorva/a-simple-strategy-to-manage-your-javascript-projects-dependencies-d413f3d0ed2f

[10] Mybridge. 2018. 25 Amazing Open Source React.js Projects for the Past Year. https://medium.mybridge.co/react-js-open-source-for-the-past-year-2018-a7c553902010

[11] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesus M. Gonzalez-Barahona. 2019. On the Impact of Outdated and Vulnerable Javascript Packages in Docker Images. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 619–623. https://doi.org/10.1109/SANER.2019.8667984