NODE.JS PROJECT ARCHITECTURE WITH SHARED DEPENDENCIES FOR MICROSERVICES

Oleh Chaplia, PhD Student, Halyna Klym, Dr.Sc., Prof.,
Lviv Polytechnic National University, Ukraine;
e-mails: halyna.i.klym@lpnu.ua

Abstract. Microservices is an architectural style in software development that involves constructing a big solution using small, self-contained services. A set of services are connected via well-defined APIs and work together like a coherent system. The application of microservices architecture spans a wide range of domains, e.g., healthcare, finance, government, military, gaming, and entertainment.

This article analyzes existing project architecture approaches for Node.js, and improves scalable project architecture for Node.js using shared dependencies. The proposed project architecture with shared module dependencies is explicitly created for Node.js microservice. Also, the article shows the results obtained from a test project that was created based on the proposed architecture.

Key words: Cloud Computing, Microservices, NodeJS, Project Architecture, Source Code, API Services

1. Introduction

Cloud computing is currently a widespread and trending topic. Many prominent and small companies are adopting cloud technologies to optimize their processes. In today's landscape, cloud technologies have become fundamental across various sectors, such as banking, finance, government, healthcare, and military. To harness cloud computing potential, businesses must transform their software web service design. Leading industry players like Netflix, Amazon, and Spotify set trends in cloud computing.

Microservices [1] is an architectural approach in software development where a solution comprises small, independently deployable services that communicate through well-defined APIs, allowing them to work together to deliver the overall application's functionality [2]. The microservices approach is especially suitable for complex and large-scale applications where flexibility, scalability, and rapid development are essential [2].

While microservices offer numerous benefits like scalability, agility, and resilience, they also demand meticulous planning, effective communication, careful orchestration, and clear project architecture [3, 4]. Any software engineer or developer should weigh these factors to determine if the benefits of microservices align with their specific needs and resources. Engineers may use an evolutionary approach to build microservices from a single service instance to many connected instances.

The journey of developing microservices is accompanied by a set of challenges. Developing microservices within a team requires careful consideration, coordination, and strategy [3, 4].

Design consistency within a project or solution scope becomes a concern when different teams work on distinct microservices, making it vital to establish design guidelines and encourage cross-team collaboration to maintain coherence. Clear project architecture and documentation a key to successful development and support of the microservice. The necessity of managing dependencies between microservices for seamless deployment and system stability calls for the use of dependency management tools and version control practices. Ensuring compatibility between evolving microservice versions is a concern that can be addressed by implementing versioning strategies, defining clear APIs, and conducting compatibility tests. Ensuring the quality of each microservice through individual testing and monitoring can lead to fragmented oversight, which can be improved by implementing continuous integration and delivery pipelines and utilizing automated testing tools, that are integrated into well-done project architecture. Most of these challenges are directly or indirectly related to the project architecture, and how it is implemented and supported within the development lifecycle [1-4].

2. Drawbacks

Node.js, based on the JavaScript programming language, is a versatile application development framework [5]. Especially conducive to microservices, Node.js excels in efficiently managing asynchronous and nonblocking I/O operations. Its lightweight runtime, scalability, extensive JavaScript ecosystem, and support for high-performance, event-driven architectures make it an ideal choice. JavaScript, a prototype-based programming language, avoids confinement to a single programming paradigm. This versatility enables it to integrate features from diverse programming languages, making it highly adaptable for contemporary development needs.

While there is no universal standard for organizing project architecture in Node.js microservices, various best practices, patterns, and recommendations exist for structuring the source code base [6-7]. The architectural approaches employed include:

1. Layered architecture – the server code is compartmentalized into distinct layers, each carrying specific responsibilities and interactions. This design enhances modularity and separation of concerns.
2. MVC-like approaches – the server application adopts an organization resembling the Model-View-Controller (MVC) pattern. It entails categorizing the application into models, views, and controllers. Models represent data structures, views handle presentation logic, and controllers oversee business logic and request management [6, 8].
3. Event-driven architecture – the server code adheres to event-driven principles, employing event handlers and emitters. This framework facilitates decoupled and asynchronous communication via events [9].
4. Plugin architecture – the server is equipped with the potential to incorporate plugins residing within the repository, introducing supplementary functionality dynamically. However, this architecture is less prevalent in Node.js applications.

Several aspects should be highlighted in connection with MVC-like patterns [6, 8]. Initially designed for earlier programming frameworks, these patterns exhibit varied implementations across languages and frameworks. They were tailored to accommodate the traditional n-tier architecture, featuring a solitary backend server API. A subset of these patterns is stateful or mandates Server-Side Rendering (SSR). In contrast, contemporary architectural solutions often opt for autonomous web applications leveraging React.js or Angular.js frameworks for user interface delivery (UI). As seen in Netflix's streaming platform, backend API services may be constructed from a network of intricate microservices. These microservices typically maintain a public API for interfacing with other web services.

3. Goal

This research aims to analyze existing project architecture approaches for Node.js and create improved scalable project architecture for Node.js microservice. The architecture should be designed to support a modular structure, minimizing dependencies between modules and using shared dependencies between Node.js services. It should facilitate the effortless addition or removal of modules for improved scalability.

4. Recommended approaches for Node.js microservices

The considered architecture must be flexible enough to accommodate evolving requirements and future enhancements without causing substantial disruptions. It should adhere to clean coding principles and embrace the separation of concerns to promote maintainability. Extensibility is critical, allowing for the seamless integration of new features and future modifications without extensive changes. Finally, comprehensive documentation is essential, offering clear guidelines for the system's structure and its various components. This study portrays the project's architecture as a dependency graph, with each node representing a component. A component corresponds to a source code file with a ".js" extension. The analysis of the project's architecture focuses solely on components and their interconnections without delving into specific business logic implementations. Every programming language has its paradigms, best ways, patterns, and approaches to the logical organization of the source code. Also, it is essential to note that code structure depends on the business domain for which the application is built. Node.js is a framework for building applications using JavaScript programming language [5]. JavaScript is a prototype-based programming language not locked to one specific programming paradigm. It supports many features from different programming languages, making it very usable in the modern world. Therefore, it can be said that JavaScript supports OOP (Object-Oriented Programming), FP (Functional Programming), and procedural programming. Object-oriented programming is a programming paradigm based on the concept of objects, which contain specific data and logic of its domain. Each object may represent some entity, its actions, and data. Classes are definitions of the objects, and objects are instances of those classes created by invoking a constructor command. Objects may have private data and provide a public API interface for other objects. A good way is to use encapsulation to hide all implementation details and provide a minimal public API interface for other users. It also helps to hold objects as an easily upgradable and replaceable component. The main fundamental things in OOP are abstraction, encapsulation, inheritance, and polymorphism. After introducing ES6 classes in JavaScript, we can describe classes more clearly, like in C++ or Java.

5. Proposed Node.js project architecture with shared modules

The foundation of the proposed project architecture rests upon layers and components. Layers serve as the logical structure for organizing components; each layer can contain one or more components. Components implement various elements such as system libraries, tools, and business logic. The module is a JavaScript file containing the component's implementation and may be imported into another file.

We try to cleanly organize source code for the Node.js microservice using layers and reduce cross-dependencies between components and modules as much as possible.

It is known that cloud solutions consisting of many microservices often use the same source code or a similar one. Similar source codes may appear when different teams must implement their solutions for the same problems. Alternatively, when communication between multiple teams is poorly organized or impossible. There are many causes why duplicated or similar source codes appear. Engineers and developers should have approaches and tools to solve their issues well.

The proposed solution also involves sharing foundational source code, which has the potential to be applied by multiple services across all teams. This shared codebase may be hosted on NPM (Node Package Manager) as one or multiple private or public packages. NPM stands for "Node Package Manager." It is a software package manager for the Node.js runtime environment. NPM allows developers to quickly discover, install, and manage external packages or libraries, also known as "packages in Node.js projects. These packages often contain reusable code modules that help streamline development by providing ready-made functionality for various tasks. Development teams should select the type of hosting. It just impacts the visibility and privacy of the packages. The other microservices only need to retrieve and use this shared source code.

The graphical representation of this layer-based arrangement is depicted in Figure 1.

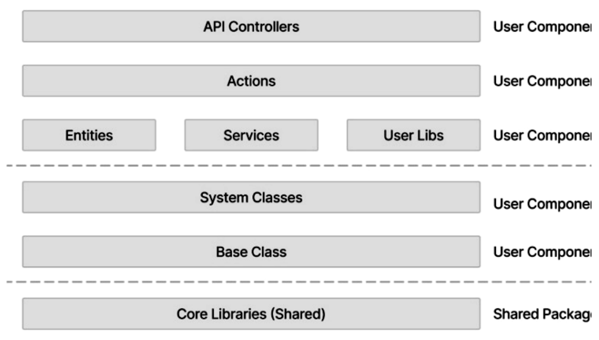


Fig. 1. Layer diagram of the components of the Node.js project architecture.

- API Controllers – functions that are attached to REST API endpoints. They import one or many Actions and execute them. Controllers also may have additional JSON validations, error handling, and logging.
- Actions – bundles business logic by combining entities, services, and libraries and providing a publicly accessible controller interface.
- Entities – maps the data from the database to the microservice. The developer can define how he wants to represent the data.
- Services – contains API calls to third-party services.
- Libs – a set of tools precisely needed for the current project. These tools are not intended to be shared between microservices.
- System Classes – Provide interface and implementation of the core functionality for Entities, Services, labs, Actions, and Controllers. These classes may have different implementations between microservices; therefore, developers can implement them as they need.
- Base Class – consolidates core libraries into one package and provides one entry point for other components. It is needed to prevent the direct import of NPM packages and reduce dependencies.
- Core Libraries – provide essential functionality for the system framework. The shared code is downloaded from NPM.

Fig. 2. represents the relations between the components based on the proposed layer diagram in Fig. 1. Arrows represent the relationship between the components.

The diagram has two types of relations. “Extends” means that the child class extends the parent class. “Imports” means that the component is imported via “require ()” or “import” methods. “#1” or “#N” means having multiple components of such type is possible. For example, “Entity #1” and “Entity #N” mean that in the current diagram, we have from 1 to N components of the class Entity. “N” can be any number. There is no upper limit, and it is based on the business requirements. It is recommended to use an object-oriented approach with JavaScript or TypeScript if possible because it provides inheritance and composition, which is very useful for organizing components.

6. Proof-of-concept implementation

A test project was created to demonstrate the practical results of the proposed Node.js project architecture. A layered structure was replicated. Also, additional items were added – ai-commands, files, security, health, and users – just for testing purposes. The application contains no business logic, external API requests, or database connections. The project structure is represented in Fig. 3. Dependency Cruiser visualizes Node.js project dependencies through import statements. All system library packages are downloaded from NPM [10]. NPM packages are stored within the “node_ modules” folder. Using NPM or other package manager for Node.js is encouraged to provide shared packages within the solution architecture. All NPM packages are connected to the Base class within the module interface. Also, it is possible to directly import NPM packages to the Base class without needing a separate interface. Anyway, developers should choose what way is better for them.

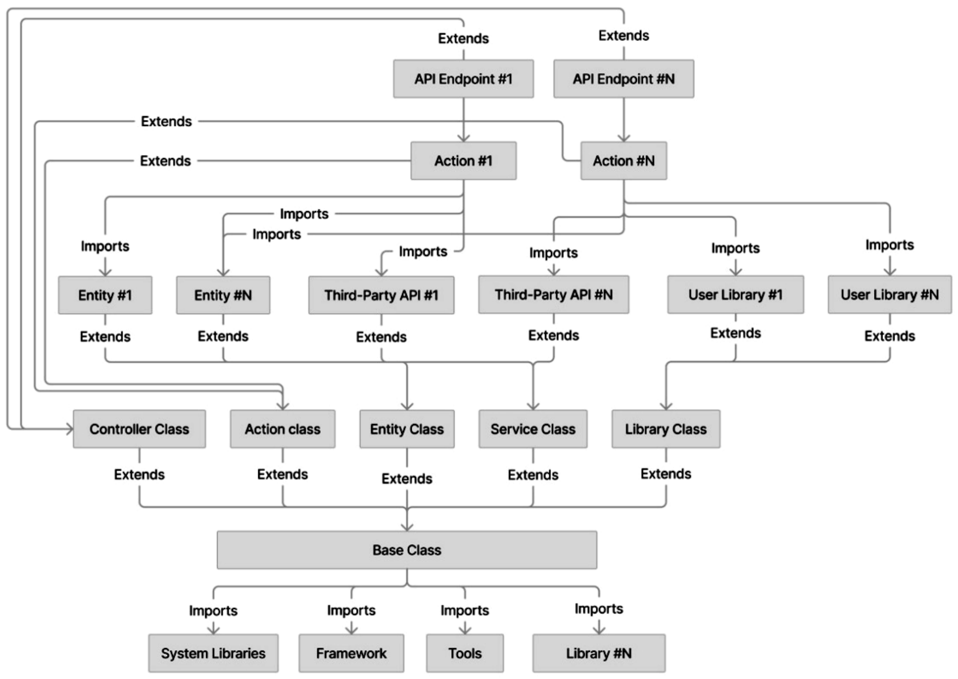
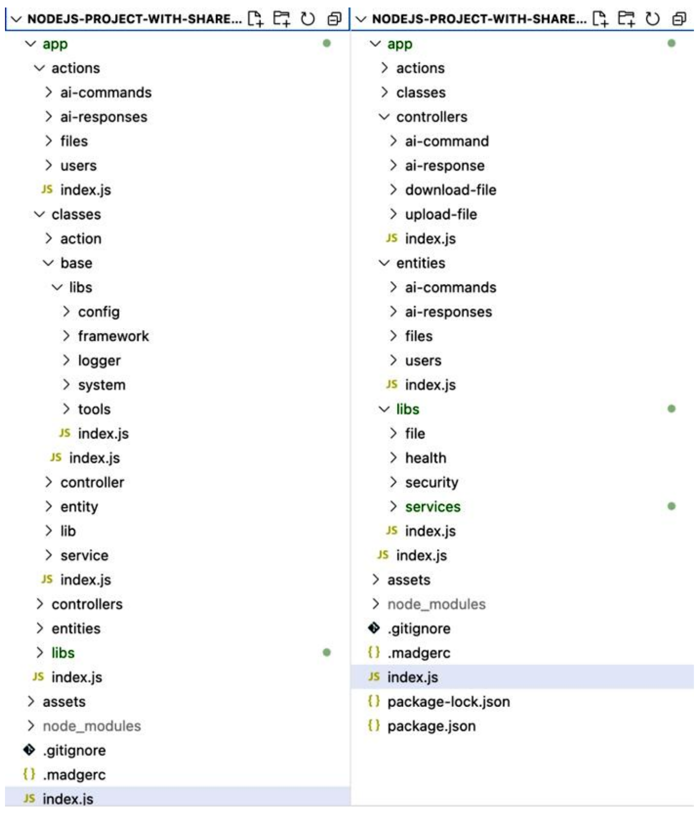


Fig. 2. Relations between components of the Node.js project architecture.



```
NODEJS-PROJECT-WITH-SHARE...
  app
    actions
      ai-commands
      ai-responses
      files
      users
      index.js
    classes
      action
      base
        libs
          config
          framework
          logger
          system
          tools
          index.js
        index.js
      controller
      entity
      lib
      service
      index.js
    controllers
    entities
    libs
    index.js
  assets
  node_modules
  .gitignore
  .madgerc
  index.js
```

```
NODEJS-PROJECT-WITH-SHARE...
  app
    actions
    classes
    controllers
      ai-command
      ai-response
      download-file
      upload-file
      index.js
    entities
      ai-commands
      ai-responses
      files
      users
      index.js
    libs
      file
      health
      security
      services
      index.js
    index.js
  assets
  node_modules
  .gitignore
  .madgerc
  index.js
  package-lock.json
  package.json
```

Fig. 3. File structure representation within IDE of the Node.js project architecture

7. Results

The outcomes were collected for further evaluation after successful implementation and testing of a proof-of-concept. A visual representation, Fig. 4, depicts a dependency graph showcasing the relationships within the implemented solution.

The visual representation of the test concept replicates the proposed architectural layers of the project solution. Action, Controller, Entity, Lib, and Service classes extend the core Base Class interface. They are accessible through the "app/classes/index.js" file. By using this file, a clear demarcation is established between system-specific classes and customized business logic.

The Core Libs layer, Base Class, and the subclasses (Action, Controller, Entity, Lib, and Service) are clearly defined within the system. The system acquires a well-defined layered arrangement that maintains loose coupling. This structural attribute persists even when new components are introduced into the system. This flexibility empowers contributors to make decisions aligned with their specific needs. This division of responsibilities can enhance developer focus, allowing them to prioritize the development of custom business logic while minimizing the requirement to support generic system tools and libraries.

The business logic crafted by users within custom Actions, Controllers, Entities, Libs, and Services embodies modularity, scalability, and flexibility. Only a specific portion of the project architecture is affected when modifications are made to these components – whether through additions, removals, updates, or deletions. This targeted adjustment minimizes overall complexity and streamlines maintaining and evolving the system.

Documenting the implemented system components is achievable through automated tools or manual documentation practices. The project's proposed architecture includes clear guidelines that assist in structuring and organizing these components.

This methodology incorporates models (entities) and controllers like MVC-style design patterns. Additionally, it introduces an additional layer called actions. Also, shared packages are attached to the libs layer within this design.

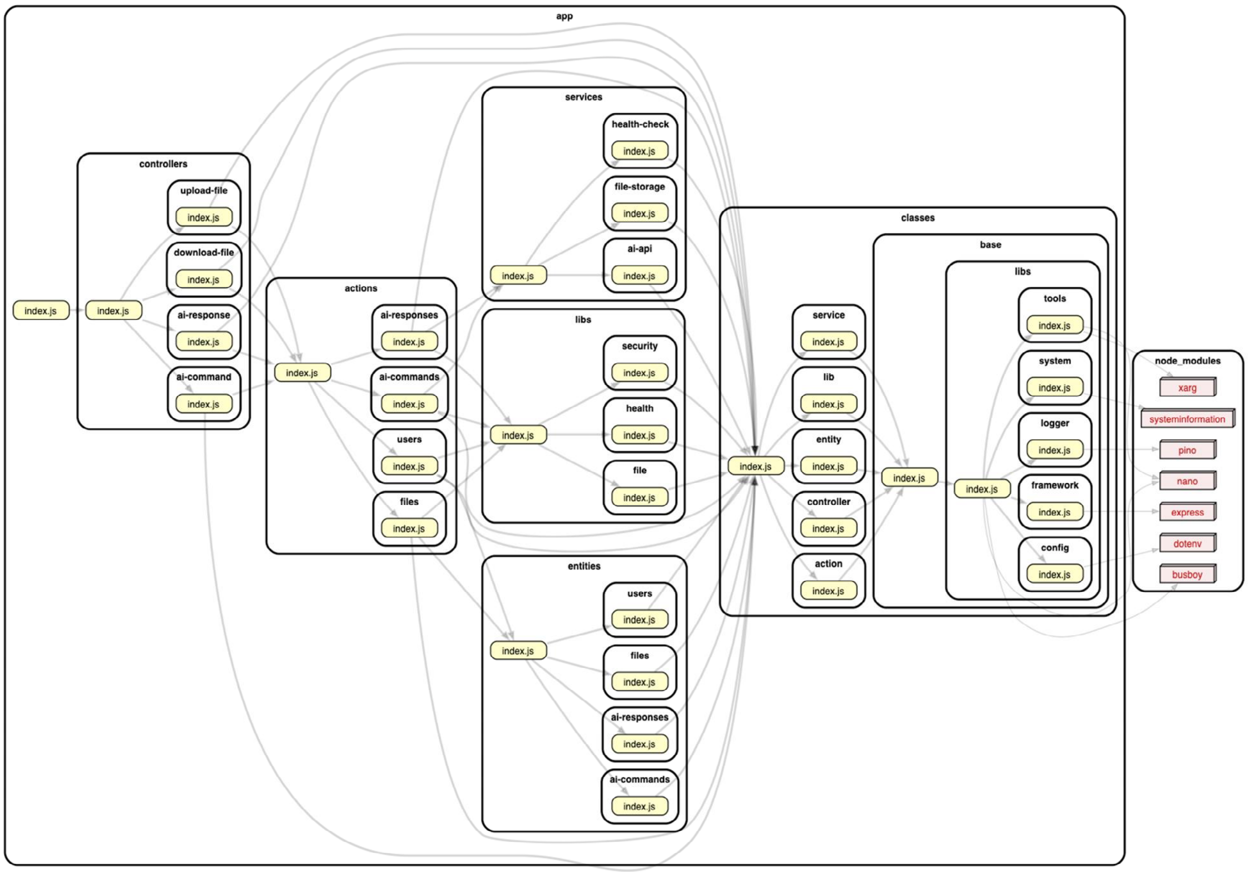


Fig. 4. Visual representation of the dependencies of project modules for Node.js project architecture generated by Dependency Cruiser.E

These actions function as composite components, intermediaries between entities and controllers to execute business logic. Consequently, the shift of business logic from controllers to actions enhances the modularity and loose coupling of the system. The services layer is responsible for interactions with external third-party services. The proposed approach emphasizes distinct layers that shield custom business logic from system libraries, tools, and classes, promoting separation.

The architecture can be adapted to accommodate database changes, third-party APIs, and external services. Its versatility makes it applicable to various programming languages, frameworks, and tools employed in microservices development.

8. Conclusions

In the suggested approach, the source code for a Node.js project's architecture can be well-defined and organized to build either a single microservice or a group of microservices. Connections between the logical layers are clear and cohesive. Shared modules with generic source code and tools (NPM packages stored inside the "node_modules" folder) may be easily added, updated, replaced, or removed within the system. Other custom components have a defined place within the source code structure according to the logical layers. System tools and business logic are placed distinctly within the proposed project architecture. This separation of components enables convenient updates, replacements, maintenance, or refinements of system or business logic components by different members of the same team or various software development teams.

9. Gratitude

The authors express their gratitude to the Department of Specialized Computer Systems and the Ministry of Education and Science of Ukraine for their support (Project for Young Scientists No 0122U000807).

10. Conflict of Interest

The authors state that there are no financial or other potential conflicts regarding this work.

References

[1] A. S. Abdelfattah, T. Cerny, Roadmap to Reasoning in Microservice Systems: A Rapid Review [J]. Applied Sciences, 13(3), 1838 (2023), DOI: 10.3390/app13031838, https://www.mdpi.com/2076-3417/13/3/1838

[2] G. Blinowski, A. Ojdowska, A. Przybyłek, Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation [J]. IEEE Access, 10, 20357-20374 (2022), DOI: 10.1109/ACCESS.2022.3152803, https://ieeexplore.ieee.org/abstract/document/9717259

[3] M. E. Gortney *et al.*, Visualizing Microservice Architecture in the Dynamic Perspective: A Systematic Mapping Study [J]. IEEE Access, 10, 119999-120012 (2022), DOI: 10.1109/ACCESS.2022.3221130, https://ieeexplore.ieee.org/abstract/document/9944666

[4] A. Baabad, H. B. Zulzalil, S. Hassan, S. B. Baharom, Software Architecture Degradation in Open Source Software: A Systematic Literature Review [J]. IEEE Access, 8, 173681-173709 (2020), DOI: 10.1109/ACCESS.2020.3024671, https://ieeexplore.ieee.org/document/9200327

[5] H. Shah, Node.Js challenges in implementation [J], Global Journal of Computer Science and Technology, 17(2), 76 (2017), https://computerresearch.org/index.php/computer/ article/download/1735/1719

[6] Aniche, M., Bavota, G., Treude, C. et al. Code smells for Model-View-Controller architectures [J]. Empir Software Eng **23**, 2121–2157 (2018). DOI: 10.1007/s10664-017-9540-2, https://link.springer.com/article/10.1007/s10664-017-9540-2

[7] F. Kaimer, P. Brune, Return of the JS: Towards a Node. js-Based Software Architecture for Combined CMS/CRM Applications [J]. Procedia Computer Science, 141, 454-459 (2018), DOI: 10.1016/j.procs.2018.10.143, https://www.sciencedirect.com/science/article/pii/S1877050918317927

[8] A. Sunardi, Suharjito, MVC Architecture: A Comparative Study Between Laravel Framework and Slim Framework in Freelancer Project Monitoring System Web Based [J]. Procedia Computer Science, 157, 134-141 (2019), DOI: 10.1016/j.procs.2019.08.150, https://www.sciencedirect.com/ science/article/pii/S1877050919310683

[9] K. Farias, L. Lazzari, Event-driven Architecture and REST Architectural Style: An Exploratory Study on Modularity [J]. Journal of Applied Research and Technology, 21(3), 338-351 (2023), DOI: 10.22201/icat.24486736e.2023.21.3.1764, https://jart.icat.unam.mx/index.php/jart/article/view/1764

[10] R. G. Kula, A. Ouni, D. M. German, K. Inoue, On the impact of micro-packages: An empirical study of the npm javascript ecosystem, arXiv preprint, arXiv:1709.04638 (2017), DOI: 10.48550/arXiv.1709.04638, https://arxiv.org/abs/ 1709. 04638